Tampereen yliopisto

Tuomas Östman

ORM-MALLIEN AIKATEHOKKUUSVERTAILU .NET-ALUSTOILLA

Informaatioteknologian ja viestinnän tiedekunta
Pro gradu -tutkielma
Maaliskuu 2020

TIIVISTELMÄ

Tuomas Östman: ORM-mallien aikatehokkuusvertailu .NET-alustoilla
Pro gradu -tutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Maaliskuu 2020

Olio-ohjelmoinnin suosion kasvaessa ja relaatiotietokantajärjestelmien käytön ollessa suosiossa, tuli ongelmaksi näiden kahden teknologian yhdistäminen. Olio-ohjelmoinnin ja relaatiotietokantojen välisiä ongelmia kuvataan yleisesti termillä Object-relational impedance mismatch. Näihin ongelmiin kehitettiin ratkaisuksi Object Relational Mapping (ORM) -malleja. ORM-mallit toimivat välitilana relaatiotietokannan ja ohjelmakoodin välillä helpottaen kehitystyötä. ORM-mallit perustoiminnaltaan hoitavat ohjelmakoodin ja relaatiotietokannan välistä yhteyttä, mutta ne sisältävät myös valmista toimintalogiikkaa, joka tuo lisää resurssivaatimuksia ohjelmalle. Tutkimuksessa tehtiin aikatehokkuusvertailu Dapper ja Entity Framework ORM-malleille käyttäen ADO.NET-ohjelmistokehystä perustason vertailukohtana. Tehokkain kolmesta oli ADO.NET, jonka jälkeen tuli Dapper ja kolmanneksi Entity Framework. Eri .NET-ohjelmistokehykset olivat tasaväkisiä, mutta .NET Core osoittautui nopeammaksi.

Avainsanat: Relaatiotietokannat, ORM (Object Relational Mapping), .NET, SQL

Sisällys

1 Johdanto

Digitalisaation aikakaudella erilaisia sovelluksia ja verkkosivuja käytetään yhä enenevissä määrin. Heinäkuussa 2019 on arvioitu maailmassa olevan noin 4,33 miljardia aktiivista internetin käyttäjää. Näistä käyttäjistä suurin osa on Aasiassa, jossa aktiivisten internetin käyttäjien määrä on yli 2,3 miljardia. Toiseksi suurin alue internetin käyttäjien määrällä mitattuna on Eurooppa hieman yli 700 miljoonalla internetin käyttäjällä. [Statista 2019a, 2019b]

Monet sovellukset ja verkkosivut käyttävät tietoa, jota ei ole varastoitu sovellukseen tai verkkosivuun itseensä. Koska tietoa voi olla todella paljon, olisi tehotonta ja epäkäytännöllistä tallentaa kaikkea käyttäjän päätelaitteelle. Tämän takia, sovellusten ja verkkosivujen toteutukset eivät usein sisällä kuin niiden toiminalle välttämätöntä tietoa. Suuri osa tiedosta on tallennettu jonnekin taustajärjestelmään, josta se haetaan käyttöön vain silloin kun sitä tarvitaan. Tämä taustajärjestelmä on usein jokin tietokantajärjestelmä.

Tietokantajärjestelmiä ja niiden toteutuksia on erilaisia ja ne voivat perustua eri tietomalleihin. Tällä hetkellä kaikista yleisimmässä käytössä ovat relaatiotietokannat (*relational database*). Db-Engines-sivuston [2019a] listauksen mukaan neljä eniten käytettyä tietokantaa ovat relaatiotietokantoja. Muita tietomalleja ovat muun muassa hierarkkinen tietomalli, verkko-, oliotietomalli.

Ohjelmistokehykset (*software framework*) ovat osa olio-ohjelmointia. Riehlen [2010] mukaan ohjelmistokehykset koostuvat valmiista luokista (*class*) ja kehyksistä (*framework*), joita voi käyttää sovelluskehityksessä ohjelmiston perusrakenteina. Ohjelmistokehykset sisältävät paljon valmista logiikkaa, mikä helpottaa sovelluskehitystyötä. Ohjelmistokehyksiä on monia erilaisia ja ne voivat olla rakennettu vain tiettyä toimintaa varten tai ne voivat olla myös laajempia kokonaisuuksia. Samasta aiheesta voi olla myös useampi erilainen ohjelmistokehys, joten kehittäjälle on tarjolla vaihtoehtoja.

Microsoftin .NET-ohjelmistokehykset ovat esimerkkejä ohjelmistokehyksestä, joka on todella kattava ja tarjoaa valmiita luokkia moneen eri käyttöön. Näitä ohjelmistokehyksiä ovat .NET Framework ja .NET Core. Näistä vanhempi on .NET Framework, jonka ensimmäinen versio .NET Framework 1.0 julkaistiin jo 13. helmikuuta 2002 ja uusin versio

.NET Framework 4.8 julkaistiin 18. huhtikuuta 2019 [Guru99, 2019]. Uudemman ohjelmistokehyksen .NET Coren ensimmäinen versio .NET Core 1.0 julkaistiin 27. kesäkuuta 2016 ja viimeisin versio .NET Core 3.0 julkaistiin 23. syyskuuta 2019 [Microsoft, 2016].

ORM-mallit (*object-relational mapping*) ovat usein ratkaisu, joka voidaan valita, kun ohjelmassa halutaan hakea ja käyttää tietokantaan tallennettua tietoa. ORM-mallit tekevät tietokantojen käyttämisestä abstraktimpaa, mikä helpottaa sovelluskehitystä. ORM-malleissa tietokannan tiedot voidaan hakea suoraan olioiksi, jolloin tiedon käyttäminen on jatkossa yksinkertaisempaa. Kun ohjelman täytyy pystyä hakemaan tietokannasta tarvittaessa suuria määriä tietoa, täytyy ORM-mallin olla mahdollisimman tehokas, jotta ohjelmassa ei esiintyisi liian suuria viiveitä.

Tässä tutkielmassa keskitytään ADO.NET:iin ja kahteen eri ORM-malliin. Näitä ovat Dapper ja Entity Framework. ADO.NET on .NET-ohjelmistokehyksien perusosa, jolla suoritetaan tietokantayhteyksiä. Dapper on pienikokoinen tietokantayhteyksiä varten toteutettu ohjelmistokirjasto .NET ohjelmistokehyksille. Entity Framework on myös osa .NET-ohjelmistokehyksiä ja se on laajalti käytössä.

Tutkielman tarkoituksena on selvittää, miten edellä mainitut ohjelmistokehykset ja ORM-mallit voivat vaikuttaa ohjelman suorituskykyyn. Tarkemmin suorituskyvyssä keskitytään aikatehokkuuteen, eli siihen, kuinka kauan aikaa ohjelman suorittamisessa kuluu. Suorituskykyyn kuuluu myös esimerkiksi resurssienkäyttö ja skaalautuvuus pienille ja isoille tietomäärille, mutta tässä tutkielmassa ei perehdytä niihin enempää ja keskitytään vain aikatehokkuuteen.

Tutkimuskysymyksinä ovat:

1. Syntyykö aikatehokkuuseroja .NET Framework ja .NET Core -ohjelmistokehyksien välillä ja jos, niin kuinka paljon?
2. Syntyykö aikatehokkuuseroja ADO.NET:n ja Dapper ja Entity Framework ORM-mallien välillä ja jos, niin kuinka paljon?
3. Onko tietomäärän kasvulla suurempi kuin kertaantuva vaikutus ohjelman suoritusaikaan?

Tutkimuskysymyksiin haetaan vastausta case-tutkimuksella. Tutkimuksessa mitataan aikatehokkuutta sekä .NET Frameworkilla toteutetulla ohjelmalla, että .NET Corella toteu-

tetulla ohjelmalla. Molemmissa on toteutettu sama toiminnallisuus vertailupohjana toimivalla ADO.NET-ohjelmistokehyksellä ja Dapper ja Entity Framework ORM-malleilla. Jokainen toteutus sisältää kolme eri tietokantatoiminnallisuutta. Toisin sanoen, erilaisia toteutuksia kustakin tietokantatoiminnallisuudesta on 24 kappaletta. Näin saadaan mitattua aikatehokkuus jokaisen ohjelmistokehyksen ja ORM-mallin yhdistelmälle erikseen.

Luvussa 2 käydään ensin läpi tietokantojen historiaa, jonka jälkeen tutustutaan relaatiotietokantoihin ja niiden teoriaan tarkemmin. Samalla käydään läpi yleisimpiä relaatiotietokantajärjestelmiä. Relaatiotietokantojen jälkeen kerrotaan pääpiirteet yleisimmästä relaatiotietokantojen kyselykielestä eli SQL-kielestä. Lopuksi listataan muutamia muita käytössä olevia tietokantamalleja.

Luvussa 3 perehdytään ORM-malleihin ja niiden teoriaan. Lisäksi tutustutaan relaatiotietokantajärjestelmien ja olio-ohjelmoinnin välisiin ongelmiin, eli *object-relational impedance mismatch* -ongelmiin. Käydään myös läpi, miten ORM-mallit vaikuttavat ohjelmistokehitykseen. Tämän jälkeen tutustutaan tarkemmin myös ADO.NET:iin, joka toimii tutkimuksen perustason vertailukohtana, ja tutkimuksen kohteena oleviin kahteen ORM-malliin Dapperiin ja Entity Frameworkiin.

Luvussa 4 aiheena on ohjelmistokehykset ja niiden teoria. Samalla käydään läpi ohjelmistokehyksien hyviä ja huonoja puolia ohjelmistokehityksessä. Seuraavaksi tutustutaan .NET-ohjelmistokehyksiin ja näistä tarkemmin kahteen ohjelmistokehykseen .NET Frameworkiin ja .NET Coreen. Viimeiseksi tarkastellaan, miten .NET Framework ja .NET Core poikkeavat toisistaan ja miltä osin ne muistuttavat toisiaan.

Luvussa 5 siirrytään tutkimukseen ja kerrotaan tutkimusmenetelmästä, sekä tarkennetaan tiedot ympäristölle, jolla tutkimus tehdään. Tämän jälkeen tarkastellaan tutkimukseen valittua tietoa ja sen ominaisuuksia. Sen jälkeen esitellään tiedon pohjalta luotua tietokantaa, jota tullaan käyttämään tutkimuksen relaatiotietokantana. Seuraavaksi tutustutaan myös ohjelman lähdekoodiin. Luvun lopuksi esitetään tutkimuksen tulokset ohjemistokehyksittäin ja ORM-malleittain.

Luvussa 6 käydään läpi viidennessä luvussa esitellyt tulokset ja analysoidaan niitä. Verrataan ORM-mallien tuloksia toisiansa ja verrokkina olevaa ADO.NET:ä vasten. Selvitetään, onko ohjelmistokehyksellä merkitystä suoritusaikoihin ja jos on, niin kuinka suuria eroja. Selvitetään myös, kuinka suuri vaikutus rivien määrällä on suoritusaikojen kasvuun.

Luvussa 7 kerrataan tutkielman aiheet ja palataan tutkimuskysymyksiin, jonka jälkeen esitetään tutkimuksen tuottamat vastaukset niille. Lopuksi kerrotaan mahdollisista jatkokehitysmahdollisuuksista tutkimukselle.

2 Tietokannat

2.1 Tietokantojen historia

Nykyään suosituimpana tietokantamallina on relaatiotietokanta, joka on ollut sitä jo useamman vuosikymmenen. Tietokantojen historia alkoi, kun tietomäärät alkoivat kasvaa suuriksi, jolloin niitä ei ollut enää järkevää pitää taltioituna manuaalisessa muodossa, esimerkiksi paperilla, mapitettuna arkistoihin. Sopivasti samaan aikaan tietokoneiden kehitys oli päässyt siihen pisteeseen, että niiden avulla oli mahdollista tallentaa yhä suurempia ja suurempia tietomääriä. Ensimmäisenä oikeana tietokantajärjestelmänä pidetään 1960-luvulla kehitettyä hierarkkista tietokantamallia (*hierarchical database model*). [Elmasri ja Navathe 2017]

Hierarkkiselle tietokantamallille ei ole olemassa alkuperäistä dokumenttia, jossa malli olisi alun perin kuvattu. Elmasri ja Navathe [1989 s.253] kirjoittavat, että hierarkkinen tietokantamalli luotiin kuvaamaan monia hierarkkisia organisaatioita, joita ilmenee maailmassa. Malli ei kuitenkaan sovi kuvaamaan ei-hierarkkisia suhteita. He listaavat hierarkkisella tietokantamallilla olevan ainakin kolme rajoittavaa tekijää, kun tietomallia suunnitellaan. Ensimmäinen rajoite on se, että vain juuritietue voi olla olemassa ilman, että sillä on vanhempitietuetta olemassa. Tämän takia lapsitietuetta ei voi viedä tietokantaan, jos sillä ei ole vanhempaa. Lapsitietueet voidaan poistaa itsenäisesti, mutta vanhemman poistaminen poistaa myös lapsien tiedot automaattisesti. Toinen rajoite on se, että jos lapsitietueella on kaksi tai useampi samantyyppinen vanhempi, täytyy lapsitietue kopioida kaikkien vanhempien alle. Kolmantena rajoitteena on se, että lapsitietue, jolla on kaksi tai useampi vanhempi on mahdollinen, jos sillä on vain yksi oikea vanhempi ja loput vanhemmat ovat virtuaalisia vanhempia.

Butterfield, Ngondi ja Kerr [2016] kuvaavat hierarkkisen tietokantamallin yhden suhde moneen -periaatteeseen perustuvana tietokantamallina. Malli koostuu siis vanhempi- ja lapsielementeistä (*parent and child element*). Lapsielementillä voi olla vain yksi vanhempielementti, mutta vanhempielementillä yksi tai monta lapsielementtiä. Juurielementti on ainoa, joka voi olla olemassa ilman suhteita vanhempi- tai lapsielementtiin. Harrington [2009 s. 393-397] kertoo hierarkkisen tietokantamallin olevan navigoinnillinen tietomalli (*navigational data model*) eli kulkureitit tietojen välillä ovat jo ennakkoon määritelty.

Tässä tietomallissa ei ole mahdollista, että lapsielementillä olisi useampi vanhempielementti. Tämä usein johtaa siihen, että todennäköisesti tietoa joudutaan tallentamaan useampaan kertaan. Tällainen tiedon monistuminen voi johtaa tiedon johdonmukaisuusongelmiin. Suosituin hierarkkinen tietokantamalli oli IBM:n kehittämä IMS, joka on kehitetty vuonna 1966. Hierarkkisesta tietomallista on periytynyt muuan muassa NP2, XML ja JSON.

Hierarkkisesta tietomallista hieman kehittyneempi malli on verkkotietomalli. Verkkotietomallin kaksi perustietorakennetta ovat tietueet (*record*) ja joukot (*set*). Jokainen tietue koostuu ryhmästä toisiinsa liittyviä tietoarvoja. Joukot koostuvat yhdestä omistajatietueesta (*owner record*) ja määrittelemättömästä määrästä jäsentietueita (*member record*). Tietueet voidaan luokitella tietuetyyppeihin (*record types*), joissa jokainen tietuetyyppi kuvaa rakenteen ryhmälle tietueita, jotka varastoivat saman tyyppistä tietoa. Joukkotyyppi (*set type*) kuvaa kahden tietuetyypin välistä yhden suhde moneen -suhdetta. Tietue voi olla jäsenenä tai omistajana joukkotyypeissä ilman määrällisiä rajoituksia. Tämä on suurin erottava tekijä verkkotietomallin ja hierarkkisen tietomallin välillä. [Elmasri ja Navathe 1989 s.281-323]

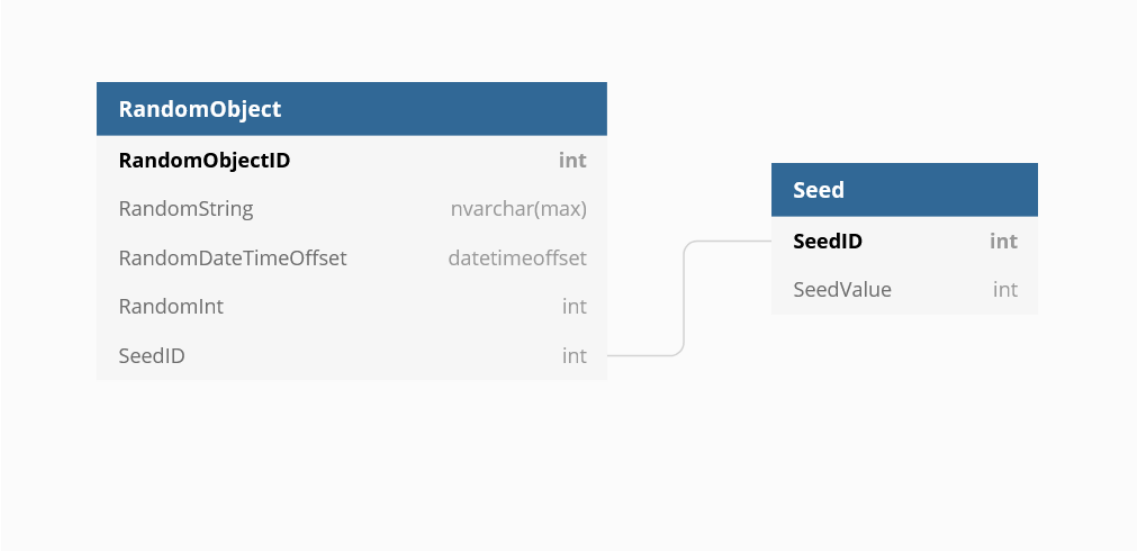
Harrington [2009 s. 397-405] esittelee verkkotietomallin myös yhden suhde moneen -periaatteeseen perustuvana tietomallina. Verkkotietomallissa ei kuitenkaan ole rajoitetta yhden lapsielementin vanhempielementtien määrään, mikä oli yksi hierarkkisen tietokantamallin suurimmista huonoista puolista. Tämä mahdollistaa useampia linkityksiä elementtien välille ja voi nopeuttaa tiedonhakua, kun voidaan rakentaa monimutkaisempia linkityksiä, jolloin tarvittava tieto voidaan löytää pienemmällä etsimisellä. Verkkotietokantamallit voivat olla hankalasti ylläpidettäviä. Esimerkiksi verkkotietokannan logiikan muuttaminen vaatii koko tietokannan varmuuskopiointia ja poistamista ennen kuin muutoksia on mahdollista tehdä.

Varhaisilla tietokantamalleilla ei ollut vielä yhteistä ohjelmointikieltä tietokannan manipulointiin, vaan ohjelmoijat pystyivät valitsemaan manipulointikielen itse. Kuten Sammet [1978] kertoo, tähän ongelmaan haettiin ratkaisua jo vuonna 1959, kun ihmiset eri aloilta kokoontuivat pohtimaan, olisiko kannattavaa kehittää uusi laitteistoriippumaton ohjelmointikieli liiketoimintaongelmiin. Myös CODASYL (*Committee on Systems Languages*) oli mukana edesauttamassa asiaa. Tuloksena oli ohjelmointikieli COBOL (*common business-oriented language*), joka on voimakkaasti englannin kieleen perustuva ohjelmointikieli. Nykyään COBOLia ei käytetä enää uusien ohjelmistojen kehittämiseen.

2.2 Relaatiotietokannat

Hierarkkinen tietokantamalli ja verkkotietokantamalli saivat suuren kilpailijan, kun Codd [1970] esitteli relaatiotietokantamallin. Relaatiotietokantamallissa siirryttiin pois navigaationallisista rakenteista ja tietoa alettiin tallentaa relaatioihin (*relation*), yleisesti relaatiotauluihin (*table*). Relaatiotaulut voivat sisältää rivejä (*row*) tai toisin sanoen tietueita ja sarakkeita (*column*). Jokainen relaatiotaulun rivi on ainutlaatuinen ja voi sisältää eri tietotyyppejä (*esimerkiksi integer, float, datetime*) eri sarakkeissa. Relaatiotauluissa voi olla myös pääavain (*primary key*), joka erittelee tiedot relaatiotaulun sisällä. Pääavaimen avulla relaatiotauluja voidaan yhdistää toisiinsa relaatioilla (*relation*). Tällaista yhdistävää saraketta kutsutaan nimellä vierasavain (*foreign key*).

Codd [1989] kuvaa relaatiomallin lisäävän tuottavuutta sekä tuotteen ohjelmoijille, että tuotteen loppukäyttäjille. Argumentit perustuvat tiedon itsenäisyyden, rakenteellisen yksinkertaisuuden ja relaatiomallissa määritellyn relaationallisen prosessoinnin ympärille. Nämä edellä mainitut asiat yksinkertaistavat ohjelmistokehitystyötä, helpottavat tietokantakutsujen luomista ja tiedon viemistä tietokantaan loppukäyttäjän päätteeltä. Tiedon itsenäisyys pitää myös ohjelmistot ja tietokannat käyttökelpoisina ja helpottaa niiden ylläpitoa organisaationallisten tai muunlaisten muutosten vaatiessa muutosta myös tietokantaan.



Kuva 1: Relaatiotietokantakaavio.

Kuvan 1 relaatiotietokantakaaviossa on kaksi taulua: RandomObject ja Seed. RandomObject-taulussa on pääavaimena RandomObjectID ja Seed-taulussa SeedID. Taulut on yhdistetty vierasavainsuhteella. RandomObject-taulun RandomSeedID-sarake viittaa Seed-taulun SeedID-sarakkeeseen. Tällä tavoin voidaan liittää RandomObject-tauluun tiedot Seed-taulusta lisäämättä itse Seed-taulun tietoa RandomObject-tauluun.

2.2.1 Yleisimmät relaatiotietokantajärjestelmät

Yleisimmässä käytössä olevia relaatiotietokantajärjestelmiä ovat Db-Engines-sivuston [2019a] tekemän listauksen mukaan Oracle, MySQL, Microsoft SQL Server ja PostgreSQL, järjestyksessä alkaen suosituimmasta. Db-Engines-sivulla [2019b] esitetään myös, että nämä kuusi relaatiotietokantajärjestelmää ovat olleet suosituimpia jo yli kuusi vuotta.

Vuonna 1980 julkaistu Oracle ja vuonna 1989 julkaistu Microsoft SQL Server ovat kaupallisia relaatiotietokantajärjestelmiä, joista on olemassa rajoitettu ilmainen versio. Vuonna 1995 julkaistu MySQL ja vuonna 1989 julkaistu PostgreSQL ovat avoimen lähdekoodin ilmaisia relaatiotietokantajärjestelmiä. [Db-Engines 2019c]

2.2.2 SQL

Relaatiotietokantamallien yksi suuri etu on hyvin standardoitu kyselykieli SQL (*structured query languge*). SQL-kielen esittelivät vuonna 1974 Chamberlin ja Boyce [1974] nimellä SEQUEL (*structured english query language*). Kieli kehitettiin alun perin IBM:n System R -tietokantajärjestelmää varten, mutta on sittemmin vakiintunut relaatiotietokantojen kyselykieleksi.

SQL rakentuu lausekkeista (*clause*), predikaateista (*predicate*), ilmaisuista (*expression*), kyselyistä (query) ja lauseista (s*tatement*). Lausekkeet ovat kyselyjen peruskomponentteja. Näitä ovat muun muassa SELECT FROM ja WHERE. SELECT-lausekkeessa valitaan mitkä sarakkeet halutaan kyselyssä ottaa mukaan. SELECT-lausekkeessa voidaan myös käyttää *-merkkiä, jos halutaan valita kaikki sarakkeet taulusta. FROM-lauseke määrittää taulun tai karteesisen tulon (*cartesian join*) ja liitosten (*join*) avulla taulut, joista tietoja halutaan hakea. Käyttämällä karteesista tuloa palautuu kaikkien rivien yhdistelmä,

mutta liitosten avulla yhdistämiselle voidaan asettaa erilaisia ehtoja. WHERE-lausekkeessa annetaan rajausehtoja tiedolle, jota halutaan palauttaa. Predikaatit ovat ehtoja, joilla kyselyä rajataan. Ilmaisut ovat käyttäjän antamia arvoja, joita verrataan predikaateissa esitettyihin muuttujiin. Kyselyt hakevat tietoa annettujen ehtojen mukaan. Lausunnot ovat kyselyitä, joilla voidaan esimerkiksi koota tietoa uusiin rakenteisiin.

Käyttämällä kuvan 1 relaatiotietokannan esimerkkiä kyselyllä:

```
SELECT *
FROM RandomObject
WHERE RandomSeedID = 1
```

saadaan haettua RandomObject-taulusta kaikki rivit, joissa RandomSeedID = 1. Edellä esitetyllä kyselyllä ei kuitenkaan saada tietoon sitä, mikä Seed-taulun SeedValue-sarakkeen arvo on, kun RandomSeedID = 1, koska se ei kuulu RandomObject-tauluun. Seed-taulun SeedValue-arvo saadaan mukaan muokkaamalla kysely muotoon:

```
SELECT *
FROM RandomObject, Seed
WHERE RandomSeedID = 1.
```

Tällöin tietokannasta palautuu myös arvot Seed-taulusta. Riippuen rivien määrästä palautuu myös rivejä, jotka eivät linkity RandomSeedID arvon perusteella. Muokkaamalla kysely muotoon:

```
SELECT *
FROM RandomObject
INNER JOIN Seed ON RandomObject.RandomSeedID = Seed.SeedID
WHERE RandomSeedID = 1
```

saadaan tietoon myös kyselyssä annetun RandomSeedID-arvon mukainen SeedValue-kentän arvo, sillä liitoksen avulla yhdistetään Seed-taulun tiedot mukaan kyselyyn, käyt-

tämällä RandomObject-taulun SeedID-saraketta ja Seed-taulun SeedID-saraketta yhdistävänä tekijänä taulujen välillä. Näin voidaan käyttää kahden taulun tietoja yhdessä kyselyssä. Kappaleessa esitetyt esimerkkikyselyt voidaan myös esittää eritavoin.

2.2.3 ACID

Tietokantatapahtumien eheys ja toimintavarmuus on otettu huomioon järjestelmissä jo 1970-luvulla ja sitä on tutkinut muun muuassa Grey [1981]. Haerder ja Reuter [1983] esittelivät käsitteen ACID, joka yleistyi käytetyimmäksi käsitteeksi. ACID on akronyymi sanoista atomisuus (*Atomicity*), eheys (*Consistency*), eristyneisyys (*Isolation*) ja pysyvyys (*Durability*). ACID on periaate, jolla taataan, että tietokanta pysyy eheänä jokaisessa tilanteessa. Atomisuudella tarkoitetaan kaikki tai ei mitään -tilannetta, jossa transaktio (*transaction*) joko suoritetaan kokonaisuudessaan loppuun tai se perutaan kokonaisuudessaan. Eheydellä tarkoitetaan sitä, että transaktio saa tehdä vain oikeanlaisia muutoksia, jotta tietokanta pysyy johdonmukaisena. Eheys on tärkeä osa myös neljännen kohdan, pysyvyyden, kannalta. Eristyneisyydellä tarkoitetaan sitä, että transaktioiden tulee olla piilotettuna muilta transaktioilta, joita ajetaan samaan aikaan. Jos näin ei olisi, transaktio ei voisi palata alkuperäiseen tilaansa. Pysyvyydellä tarkoitetaan sitä, että transaktion valmistuttua, järjestelmän tulee taata, että transaktion muutokset ovat pysyviä. Käyttäjän tulee olla varma, että transaktio on onnistunut. Siksi jokainen loppuun saatettu transaktio on perusarvoltaan aina onnistunut.

2.3 Muita tietokantamalleja

Aikaisemmin mainitut hierarkkinen tietokantamalli, sekä verkko- ja relaatiotietokantamallit eivät ole ainoat tietokantamallit. Muita tietokantamalleja ovat muun muassa olio- (*object-oriented database*), dokumentti- (*document-oriented database*), sarakeperhe- (*column family database*) ja avain-arvotietokannat (*key-value database*). Näitä tietokantoja kutsutaan yhteisellä nimityksellä NoSQL (*Not Only SQL*, eli ei vain SQL), joka kuvastaa sitä, että nämä tietokantatoteutukset eivät käytä SQL-kyselykieltä tietokannan manipulointiin. Han *et al.* [2011] kertoo, että NoSQL-tietokantojen hyviä puolia ovat muun muassa nopea tiedonluku ja -kirjoitus, laajentamisen helppous ja taloudellisuus.

Oliotietokantojen määrittelyä edistivät suuresti Cattell *et al.* [1994], esittelemällä ODGM-määrittelyn. Atkinson *et al.* [1990] esittelevät kattavasti pohjan oliotietokannoille. Oliotietokannat muistuttavat läheisesti olio-ohjelmointia ja sen takia oliotietokantojen uskottiin syrjäyttävän relaatiotietokannat, mutta niin ei ole vielä käynyt. Oliotietokannoissa tieto tallennetaan käyttämällä olioita ja niiden sisältämiä metodeja. Esimerkiksi yksi oliotietokantajärjestelmä on O2 [Deux 1991].

Han et al. [2011] mukaan avain-arvo-tietomalleissa jokaisella arvolla on aina avain, jolla arvon voi löytää. Avain-arvo-tietomalleissa kyselynopeus on suurempi kuin relaatiotietomallissa ja rakenne on myös yksinkertaisempi. Avain-arvo-tietomalli tukee massatallennusta ja suurta samanaikaisuutta.

Dokumenttitietokannoissa tieto voidaan tallentaa samankaltaisesti kuin avain-arvo-tietomallissa, eli arvolla on jokin avain, jolla se voidaan löytää. Dokumenttitietokannat ovat yleensä tallennettu XML (*extensible markup language*) tai JSON (*javascript object notation*) -muotoon. Dokumenttitietomalleissa arvolle voidaan asettaa myös toinen indeksi, jota ei tueta avain-arvo-tietomallissa. [Han *et al.* 2011]

3 ORM

3.1 Object-relational impedance mismatch

Ireland *et al.* [2009a] luokittelevat *Object-relational impedance mismatch* -ongelmia ja käyvät niitä läpi heidän esittämänsä nelitasoisen kehyksen avulla. Näitä tasoja ovat paradigma/malli (*paradigm*), kieli (*language*), kaavio (*schema*) ja ilmentymä (*instance*). Paradigmaongelmia ovat kaksi eri toiminnallista lähestymistapaa, joista toinen käsittelee toisiinsa liittyvien olioiden verkostoja ja toinen relaatioiden joukkoja. Kieliongelmiin kuuluu tietokantakielien ja ohjelmointikielien tietorakenteiden ja -tyyppien erilaisuus. Esimerkiksi olio-ohjelmoinnin yksi perusrakenne on luokat, joille ei löydy vastaavuutta SQL-kielestä. Kaavio-ongelma on esimerkiksi tiedon kartoittaminen tietomallista olioksi. Ilmentymäongelmiin kuuluu esimerkiksi tiedon ja rakenteen eheyden pysyvyys.

Object-relational impedance mismatch -ongelmiin ratkaisuksi kehitettiin erilaisia ORM-malleja, jotka ratkovat *Object-relational impedance mismacth* -ongelmia [Ireland *et al.* 2009b]. Tämä on yksi syy siihen, että oliotietomalli ei koskaan yleistynyt yleisemmäksi kuin relaatiotietomalli, vaikka niin ennustettiin.

3.2 ORM-mallit

ORM on akronyymi sanoista Object-Relational Mapping. ORM-mallien on tarkoitus toimia välikerroksena ohjelman ja tietokannan välillä. Cabibbo ja Carosi [2005] kuvaavat ORM-malleja kehyksinä, jotka varastoivat ja hakevat pysyviä olioita. He kertovat myös kehyksien tehtävänä olevan pitää huolta yhteyksistä olioiden ja relaatiotietokannan välillä. O'Neil [2008] esittää ORM-mallien toimivan siltana tietokantojen taulukkotiedolle ja ohjelmien olioille. Hän kertoo tämän olevan myös suotuisampaa ohjelmoijille, sillä he voivat työskennellä pysyvän tiedon kanssa, eikä heidän tarvitse käyttää SQL-kieltä suoraan tiedon hakuun. ORM-mallit pyrkivät myös tuomaan ratkaisun *Object-relational impedance mismatch* -ongelmiin.

ORM-malleja on nykyään saatavilla monelle eri ohjelmointikielelle ja usein on monta eri vaihtoehtoa. Suurin osa ORM-malleista on ilmaisia ja monet mallit ovat myös vapaan lähdekoodin toteutuksia, esimerkiksi .NET-ohjelmistokehyksille Dapper ja NHibernate.

Myös suljetun lähdekoodin toteutuksia on olemassa, kuten .NET-ohjelmistokehyksille Entity Framework.

ORM-mallit yleisesti muuntavat relaatiotietokannasta saadun tiedon olio-ohjelmointikielelle sopivaan muotoon. Esimerkiksi kuvan 1 RandomObject-taulu Entity Frameworkin muodostaman C#-kielisen luokan muoto on esimerkin 1 mukainen.

```
public partial class RandomObject
{
    public int RandomObjectID { get; set; }
    public string RandomString { get; set; }
    public DateTimeOffset RandomDateTimeOffset{ get;
                                                set; }
    public int RandomSeedId { get; set; }
    public int RandomInt { get; set; }

    public virtual Seed Seed { get; set; }
}
```

Esimerkki 1: RandomObject-olio C#-ohjelmointikielellä.

Entity Framework luo luokan relaatiotietokannan perusteella. Luokalle luodaan avainsanat *public*, *partial* ja *class. Class*, eli luokka, kertoo RandomObjectin tarkoittavan luokkaa, kun taas *public*, eli yleinen, kertoo RandomObject-luokan olevan käytettävissä kaikkialla ohjelmakoodissa. *Partial*, eli osittainen, tarkoittaa sitä, että RandomObject-luokka voi sisältää ohjelmakoodia muuallakin ohjelmakoodissa. Jokaista relaatiotaulun saraketta varten on luotu yksi ominaisuus (*property*). Jokaiselle ominaisuudelle on myös asetettu public-avainsana ja arvon hakemista ja syöttämistä varten get- ja set-funktiot.

RandomObject-taulu ja -luokka muistuttavat paljon toisiaan. Taululla ja siitä luodulla luokalla on sama nimi, kuten myös taulun sarakkeilla ja luokan attribuuteilla on samat nimet. Eroavaisuuksia näkyy vasta, kun tarkastellaan attribuuttien tietotyyppejä. Int- ja DateTimeOffset-tietotyypeille löytyy vastaavuudet sekä tietokannasta että ohjelmointikielestä, mutta RandomString-sarakkeen tietotyypille nvarchar(max):lle ei löydy vastaavuutta C#-kielestä. Nvarchar(max) kuvaa vaihtuvamittaista unicode-merkkijonoa, jolle ei ole asetettu maksimipituutta, sillä sulkujen sisällä on sana *max*, eikä numeroarvoa, joka määrittäisi merkkijonon maksimipituuden. Entity Framework muuntaa attribuutin tietotyypiksi merkkijonoa vastaavan tietotyypin *string*.

3.3 Miten ORM-mallit vaikuttavat ohjelmistokehitykseen?

ORM-mallit helpottavat ohjelmistokehitystä monilla eri tavoilla. ORM-malleja käytettäessä ei yleensä tarvitse käyttää SQL-kieltä. Eli ORM-malleja käytettäessä ei välttämättä tarvitse käyttää montaa eri ohjelmointikieltä sovelluksen toteutukseen, kun tietokannan ja sovelluksen välinen kommunikointi hoidetaan ORM-mallin avulla. ORM-malli voi myös tuottaa tehokkaampia kyselyitä tietokantaan kuin ohjelmoija, joka ei tunne riittävän hyvin SQL-kieltä. ORM-mallit usein abstrahoivat relaatiotietokantajärjestelmää siten, että eri järjestelmästä toiseen siirtyminen on yksinkertaisempaa. ORM-malleissa voi myös olla paljon uusia ominaisuuksia, riippuen ORM-mallista.

ORM-mallit eivät kuitenkaan tuo pelkästään positiivisia vaikutuksia ohjelmistokehitykseen. Vaikka ORM-mallin käyttö voi olla yksinkertaista, sen käyttöönotto ja kokoonpaneminen voi olla hankalaa. Pelkän SQL:n käyttö voi olla myös huomattavasti tehokkaampaa, mutta tämä vaatii ohjelmoijalta tietämystä SQL:stä. Kuten SQL-kieli, myös ORM-mallin käyttö vaatii aina opiskelua. ORM-mallien käyttö voi myös vieraannuttaa ohjelmoijan SQL:stä, vaikka aina olisi syytä ymmärtää myös taustaa siitä, miten ORM-mallit toimivat pohjimmiltaan tietokantaa vasten.

3.4 ADO.NET

ADO.NET on Microsoftin kehittämä teknologia, jolla voidaan kommunikoida tietolähteen kanssa. Tietolähteitä voivat olla esimerkiksi Microsoft SQL Server ja XML [Microsoft 2017]. ADO.NET on osa .NET-ohjelmistokehyksiä ja se on alun perin julkaistu .NET Framework 2.0:n mukana vuonna 2006. ADO.NET on jaettu useampaan eri osaan, joista jokaista voi käyttää erikseen tai kaikkia yhdessä. ADO.NET ei siis ole ORM-malli, vaan .NET-ohjelmistokehyksien mukana tuleva pienempi ohjelmistokehys, jota käytetään tietokantayhteyksiin ja tietokantatiedon manipulointiin. ORM-mallit käyttävät ADO.NET:ä tietokannan manipulointiin, joten ADO.NET on olennainen osa myös ORM-malleja.

ADO.NET on usein nopein ja tehokkain tapa käyttää tietokantoja .NET-ohjelmistokehyksissä, sillä ORM-mallit on rakennettu ADO.NETin komponenttien päälle. ADO.NET:n käyttö ei kuitenkaan ole niin käyttäjäystävällistä kuin ORM-mallien, kuten Entity Frameworkin käyttö. ADO.NET:ssä käyttäjä joutuu vielä itse kirjoittamaan SQL-kyselyt, sillä ADO.NET ei sisällä implementaatioita SQL-kyselyistä.

```
var seedID = 0;
int seed;
using (var connection =
       new SqlConnection(ConnectionString))
{
    using (SqlCommand cmd =
           new SqlCommand("SELECT SeedValue FROM Seed WHERE
                           SeedID = @ID"
           , connection))
    {
        cmd.Parameters.Add("@ID", SqlDbType.Int).Value = seedID;
        connection.Open();
        SqlDataReader reader = cmd.ExecuteReader();
        while (reader.Read())
        {
        seed = (int)reader["SeedValue"];
        connection.Close();
    }
}
```

Esimerkki 2: ADO.Net kysely C#-ohjelmointikielellä.

Esimerkin 2 mukaan nähdään, miten kuvan 1 mukaiseen relaatiotietokantaan muodostetaan SQL-kysely ADO.NET:n avulla. Esimerkki 2 vastaa SQL-kyselyä SELECT SeedValue FROM Seed WHERE SeedID = 0. Ensimmäisenä alustetaan uusi SqlConnection-luokka yhteysmerkkijonolla (*connection string*), joka kertoo yhteystiedot tietokantaan, jonne halutaan ottaa yhteys. Seuraavaksi alustetaan itse kysely uuteen SqlCommand-luokkaan ja kysely on SELECT SeedValue FROM Seed WHERE SeedID = @ID. Esimerkissä 2 ”@ID” kuvaa tässä kyselyssä kyseistä Seed-taulun SeedID-arvoa, joka annetaan parametrina, jonka avulla saadaan valittua oikea tieto relaatiotaulusta. Esimerkissä SeedID:n arvo on 0. Seuraavaksi avataan yhteys tietokantaan, suoritetaan kysely ja luetaan kyselyn tulos muuttujaan seed, jonka avulla tulosta voidaan käyttää muualla ohjelmakoodissa.

Esimerkistä huomataan myös, että ADO.NET:n käyttö vaatii käyttäjältä SQL-kielen osaamista. ADO.NET:n käyttöön ei siis riitä, että käyttäjä osaa vain C#-kielen käytön.

3.5 Tutkimuksen ORM-mallit

3.5.1 Dapper

Dapper on avoimeen lähdekoodin perustuva hyvin kevyt ORM-malli. Dapperin kehityksen ovat aloittaneet tunnetun ohjelmointipainotteisen sivuston StackExchange.com:in takana olevat ihmiset, mutta tällä hetkellä Dapperia on kehittänyt jo yli 150 ihmistä [GitHub, 2019]. Dapperin ensimmäinen versio 1.0.0 julkaistiin 14. huhtikuuta vuonna 2011 ja uusin versio 2.0.30 on julkaistu 27. syyskuuta vuonna 2019 [NuGet 2019].

Dapper on hyvin kevyt toteutus ORM-mallista. Dapper lisää ominaisuuksia .NET Frameworkissa jo olevaan IConnection-rajapintaan (*interface*). Dapperilla voi tehdä kyselyitä, jotka palauttavat tietoa ja kartoittavat (*map*) tulokset vahvasti tyypitettyyn listaan (strongly typed list) tai listaan dynaamisia olioita (dynamic object). Dapperilla voi myös tehdä kyselyitä, joilla ei ole paluuarvoa. Dapper ei kuitenkaan kartoita koko tietokantaa valmiiksi ohjelmakoodiin, vaan käyttäjän on itse kirjoitettava oliot ja luokat valmiiksi ja annettava ne Dapperille, joka sen myötä osaa kartoittaa oikeat tiedot relaatiotietokannasta ohjelman omiin luokkiin. [GitHub 2019]

```
var seedID = 0;
var seed = 0;

using (var conn = new SqlConnection(ConnectionString))
{
    seed = conn.Query<int>("SELECT SeedValue FROM Seed
    WHERE SeedID = @ID",
    new { ID = seedID }).FirstOrDefault();
}
```

Esimerkki 3: Dapper kysely C#-ohjelmointikielellä.

Esimerkki 3 esittää saman kyselyn kuin esimerkki 2, mutta käyttämällä Dapperia. Dapperia käytettäessä alustetaan uusi tietokantayhteys samalla tavalla kuin ADO.NET:ä käyttämälläkin, eli alustamalla uusi SqlConnection-luokka tietokantaan osoittavalla yhteysmerkkijonolla. Dapperin omalle Query-metodille annetaan sama SQL-kysely kuin ADO.NET:llekin: SELECT SeedValue FROM Seed WHERE SeedID = @ID. ”@ID”

arvoksi annetaan 0, kuten myös ADO.NET toteutuksessakin. Tämä palauttaa tietokannasta suoraan int-tyyppisen seed-arvon ilman, että kyselyä tarvitsee käydä erikseen läpi, niin kuin ADO.NET:n vastaavassa toteutuksessa.

3.5.2 Entity Framework

Entity Framework on Microsoftin ADO.NET-tietokantayhteysohjelmistokehyksen päälle kehitetty ORM-malli. Entity Frameworkin ensimmäinen versio EF 3.5 julkaistiin vuonna 2008 osana .NET Framework 3.5 SP1:stä ja Visual Studio 2008 SP1:stä. Alkaen versiosta EF4.1 Entity Frameworkia on jaettu NuGet.org-sivuston kautta nuget-pakettina ja on nykyisin yksi suosituimmista paketeista kyseisellä sivustolla. Versio EF 6:sta alkaen Entity Frameworkista on tullut avoimen lähdekoodin projekti, mikä auttoi nopeuttamaan Entity Frameworkin kehitystä. Uusin versio Entity Frameworkista on EF 6.3.0. Kesäkuussa 2016 julkaistiin EF Core 1.0, joka pohjautuu uuteen ohjelmakoodipohjaan ja on suunniteltu kevyemmäksi ja laajennettavammaksi kuin alkuperäinen Entity Framework. EF 6 pysyy kuitenkin ylläpidettävänä sekä avoimen lähdekoodin projektina, että tuettuna Microsoft-tuotteena, vaikka uusia ominaisuuksia ei tällä hetkellä suunnitella. [Microsoft, 2019a]

Adya *et al.* [2007] kuvaavat Entity Frameworkia alustana, joka merkittävästi vähentää *object-relational impedance mismatch* -ongelmia ohjelmille ja tietokeskeisille palveluille. Entity Framework eroaa vastaavista ratkaisuista ja ORM-malleista merkittävästi neljällä tavalla.

Adya *et al.* [2007] kertoo ensimmäiseksi erottavaksi tekijäksi sen, että Entity Framework määrittelee rikkaan käsitteellisen datamallin, *Entity Data Modelin* (EDM). Samalla määritellään myös tiedonmanipulointikieli, *Entity SQL,* joka toimii luodun EDM:n ilmentymissä (*instance*). Entity Framework määrittelee myös oliopalvelu-kerroksen (*object services layer*), joka tuottaa ORM-mukaisen toiminnallisuuden. Microsoft [2012] kertoo EDM:n olevan *Entity-Relationship model* eli ER-kaavio. ER-mallissa entiteetit (*entity*) ovat entiteettityyppien (*entity type*) ilmentymiä, jotka ovat rakenteellisia ja niillä on jokin avain. Tällaisia entiteettityyppejä voivat olla esimerkiksi Työnantaja ja Työntekijä, joiden avaimia voisivat olla TyönantajaID ja TyöntekijäID. Microsoft kuvaa suhteita (*relationship*) suhdetyyppien (*relationship type*) ilmentyminä, jotka ovat kahden tai useamman entiteettityypin assosiaatioita.

Toinen erottava tekijä Adya *et al.* [2007] mukaan on se, että ohjelman ajoversio (*runtime*) tekee EDM:n pysyväksi ja samalla ohjelman ajoversio sisältää kartoitusmoottorin (*mapping engine*), joka tukee kyselyitä, päivityksiä ja tehokasta kaksisuuntaista kartoitusta EDM:n ja relaatiotietokannan välillä.

Kolmas erottava tekijä on Adya *et al.* [2007] mukaan se, että ohjelmia ja palveluita voidaan kehittää suoraan joko arvoperustaista käsitteellistä mallia vasten tai ohjelmointikielikohtaisia olioita vasten, jotka on rakennettu käsitteellisen abstraktion päälle.

Adya *et al.* [2007] esittelee viimeiseksi tekijäksi Microsoftin *Language integrated queryn,* eli LINQ:n. LINQ laajentaa ohjelmointikieltä tukemaan kyselyjä, mikä vähentää ja tietyissä tilanteissa myös poistaa kokonaan *object-relational impedance mismatch* -ongelmat.

```
var seedID = 0;
var seed = 0;

using (var db = new ThesisEntities())
{
    seed = (from sd in db.Seed
            where sd.SeedID == seedID
            select sd.SeedValue).FirstOrDefault();
}
```

Esimerkki 4: Entity Framework kysely C#-ohjelmointikielellä.

Esimerkissä 4 esitetään sama kysely kuin esimerkeissä 2 ja 3, mutta toteutettuna Entity Frameworkin avulla. Esimerkkiä varten on jo valmiiksi luotu EDM kuvan 1 mukaisesta relaatiotietokannasta. Entity Framework ei tarvitse SQL-kieltä, vaan se käyttää LINQ:a, joka syntaksiltaan muistuttaa paljon SQL-kieltä, sillä LINQ:ssa käytetään paljon samoja avainsanoja kuin SQL-kielessä, esimerkiksi avainsanat *from*, *where* ja *select*. LINQ-syntaksin mukaisesti avainsanat menevät kuitenkin eri järjestyksessä.

Esimerkissä 4 alustetaan ensin muuttujat seedID ja seed. SeedID tarkoittaa halutun siemenluvun Seed-taulun SeedID-arvoa ja seed on muuttuja tietokannasta palautetulle siemenluvulle. Sen jälkeen alustetaan yhteys tietokantaan luomalla uusi ilmentymä ThesisEntities-luokasta. ThesisEntities-luokka sisältää valmiiksi tietokannan yhteysmerkkijonon ja myös kuvauksen molemmista tietokannassa olevista tietokantatauluista. Nyt voidaan käsitellä tietokantaa samalla tavalla kuin muitakin ohjelmassa esiintyviä

yleisiä luokkia. Kyselyssä db.SeedID-oliosta alustetaan muuttuja sd. Sd-muuttujan avulla voidaan asettaa ehtoja kyselylle *where*-avainsanan avulla, kuten myös valita palautusarvot *select*-avainsanan avulla. Esimerkissä 4 halutaan tietokannasta palautuvan vain siemenluvun arvo, eikä kokonaista Seed-oliota, minkä vuoksi select-avainsanan jälkeen valitaan sd.SeedValue. LINQ palauttaa tässä tilanteessa iteroitavan IEnumerable-rajapinnan mukaisen olion, jolloin täytyy kutsua IEnumerable-rajapinnan funktiota FirstOrDefault, joka palauttaa iteraation ensimmäisen tuloksen tai vakioarvon.

4 Ohjelmistokehykset

4.1 Yleistä teoriaa

Fayad ja Schmidt [1997] tiivistävät ohjelmistokehyksien olevan uudelleenkäytettäviä, osittain valmiita sovelluksia, joita täytyy kehittää, jotta saadaan tuotettua tarpeen vaatimia sovelluksia. Ohjelmistokehykset on usein tarkoitettu hoitamaan yhtä tiettyä osaa toimintalogiikasta (esim. tiedon prosessointi) tai sovelluksen tuotealueesta (esim. käyttöliittymä).

Markiewicz ja Lucena [2001] esittävät ohjelmistokehyksien olevan sovellusgeneraattoreita (*software generators*), joiden ominaisuudet ovat suoraan verrannollisia ohjelmistokehyksen spesifiin sovellusalaan. Ohjelmistokehykset toimivat siis sovellusten generoinnin pohjana, sillä ne sisältävät paljon sovellusalaan sopivaa valmista toteutusta. Ohjelmistokehykset eivät sisällä ajettavaa ohjelmaa, vaan kehittäjän täytyy luoda ohjelmistokehyksestä uusi ilmentymä (*instance*), joka käyttää ohjelmistokehystä luodakseen toimintalogiikan ja ajettavan ohjelman.

Ohjelmistokehykset ovat avainasemassa, kun kehitetään ohjelmistojärjestelmiä. Ohjelmistokehykset yleensä sisältävät isompia ohjelmistokokonaisuuksia valmiiksi toteutettuna ja pienempiä ohjelmistokokonaisuuksia kehitettäessä voidaan käyttää ohjelmistokehykseen toteutettuja toimintoja. [Riehle, 2010]

Bosch *et al.* [2000] kertovat, että ohjelmistokehyksen tulee tarjota toiminnallisuus koko sovelluksen toiminta-alalle, kun taas ohjelmistokehyksen avulla luodun sovelluksen tulee tarjota sovelluksen vaatima toimintalogiikka.

4.2 Hyödyt ja haitat

Fayad ja Schmidt [1997] esittävät, että ohjelmistokehykset voivat merkittävästi parantaa sovelluksen laatua ja vähentää sovelluksen kehittämiseen vaadittavaa panostusta. He tunnistavat myös useita haasteita ohjelmistokehyksien kanssa. Ohjelmistokehyksien kehittäminen tietylle sovellusalalle voi olla hyvin hankalaa ja niiden käyttämisen oppimiseen voi mennä paljon aikaa. He esittävät esimerkiksi, että graafisen käyttöliittymän luomiseen kehitetyn ohjelmistokehyksen tehokkaan käytön oppii usein vasta 6–12 kuukaudessa,

riippuen kehittäjän aiemmasta kokemuksesta. Sovellusten vaatimukset voivat myös usein muuttua, jolloin tarpeet ohjelmistokehyksellekin voivat muuttua.

Bosch *et al.* [2000] esittävät kolme ongelmaa ohjelmistokehyksien käytössä. Ensimmäinen ongelma on se, että voi olla haastavaa valita sopiva ohjelmistokehys kyseiselle sovellusalalle ilman, että käyttää merkittävästi aikaa erilaisten ohjelmistokehyksien tutkimiseen ja niihin tutustumiseen. Toinen ongelma on se, että ohjelmistokehyksen ymmärtäminen voi olla hankalaa, mikä voi vaikeuttaa sovelluksen kehittämiseen vaadittavan ajan arvioimista. Voi myös olla hankalaa nähdä ennalta, tukeeko ohjelmistokehys kaikkia sovelluksen vaatimuksia. Kolmas ongelma on ohjelman *debuggauksen* eli virheen etsinnän hankaloituminen, sillä usein kehittäjä ei tiedä tarkalleen, mitä ohjelmistokehyksen sisällä tapahtuu.

Riehle [2010] esittelee neljä ongelmaa kehitettäessä sovelluksia ohjelmistokehyksien avulla. Ensimmäinen ongelma on luokkien monimutkaisuus (*class complexity*). Luokat määrittelevät olioiden käyttäytymistä omissa ilmentymissään, jolloin yksi luokan rajapinta ei ole välttämättä riittävä, vaan tarvitaan parempia tapoja kuvaamaan olioiden eri ominaisuuksia. Toinen ongelma on täydentävä keskittyminen luokkiin ja niiden yhteistyöhön (*complementary focus on classes and collaborations*). Ohjelmistokehyksissä luokat toimivat yhteen toistensa kanssa, jolloin käyttäjälle täytyy olla selkeää, miten ne toimivat keskenään yhteen. Kolmas ongelma on olioiden yhteistyön monimutkaisuus (*object collaboration complexity*). Olioiden yhteistyö ohjelmistokehyksen sisällä ja ohjelmistokehyksen avulla kehitettyjen luokkien välillä voi olla monimutkaista. Neljäs ja viimeinen eritelty ongelma on ohjelmistokehyksien käytön hankaluus (*difficulties of using a framework*). On helppoa käyttää ohjelmistokehystä siten, miten ohjelmistokehyksen kehittäjät eivät ole ajatelleet, mikä voi johtaa ongelmiin lopputuotteen ja loppukäyttäjien kanssa.

4.3 .NET

.NET on ilmainen avoimen lähdekoodin kehitysalusta monille erilaisille ohjelmistoille. .NET-alustaa voi käyttää esimerkiksi mobiili-, työpöytä- ja peliohjelmistojen kehitykseen. .NET-alustaa käytettäessä voidaan käyttää useampaa ohjelmointikieltä, joita ovat C#, F# ja Visual Basic. .NET mahdollistaa myös erilaisten tekstinkäsittelyohjelmistojen käytön, kuten myös erilaisten ohjelmistopakettien käytön. .NET-alustasta on olemassa

useampi erilainen toteutus, joita ovat muun muassa .NET Framework, .NET Core ja Xamarin/Mono. .NET Frameworkilla voi toteuttaa esimerkiksi internet-sivustoja ja Windows-ohjelmia. .NET Corella voi toteuttaa myös mobiilisovelluksia ja sovelluksia Linux- ja macOS-käyttöjärjestelmille. Xamarin/Mono on .NET-alustan toteutus, jolla voi toteuttaa sovelluksia kaikille suurimmille mobiilikäyttöjärjestelmille. [Microsoft 2019b]

4.3.1 .NET Framework

.NET Framework on Microsoftin kehittämä ohjelmistokehys, jonka avulla voi kehittää ohjelmistoja Windows-käyttöjärjestelmälle. .NET Framework on ensimmäinen toteutus Microsoftin .NET-alustasta ja sillä voi toteuttaa esimerkiksi internetsivuja, palveluita ja komentorivisovelluksia Windows- ja Windows Server -käyttöjärjestelmille. [Microsoft 2019c]

Kaksi pääosaa, joista .NET Framework koostuu, ovat: *Common Language Runtime* (*CLR*) eli ohjelmanajoympäristö ja *Class Library* eli luokkakirjasto. C#-, F#- tai Visual Basic -ohjelmointikielellä kirjoitetut .NET Framework -sovellukset käännetään välikielimuotoon, eli *Common Intermediate Language* (*CIL*) -muotoon. Edellä suoritettu käännös tallennetaan joko DLL- tai EXE-tiedostomuotoon. Sovellusta ajettaessa CLR käyttää välikielimuotoisia tiedostoja ja käyttää ajonaikaista kääntäjää (*Just-in-time compiler*) kääntämään välikielimuotoinen ohjelmakoodi konekieliseksi koodiksi, jota voidaan ajaa sillä arkkitehtuurilla, millä tietokone toimii. [Microsoft 2019c]

.NET Framework tarjoaa kehittäjälle paljon ominaisuuksia, joita ovat muun muassa muistin hallinta, kattava luokkakirjasto, valmiit kehitysohjelmistokehykset ja tuki useammalle ohjelmointikielelle. Monet ohjelmointikielet vaativat kehittäjältä muistinhallintaa, mutta .NET Frameworkia käytettäessä CLR hoitaa muistinhallinnan kehittäjän puolesta. Kehittäjän ei itse tarvitse ohjelmoida matalan tason toteutuksia, sillä hän voi käyttää .NET Frameworkin kattavaa luokkakirjastoa ja sen sisältämiä tyyppejä. .NET Framework tarjoaa myös usean kehitysohjelmistokehyksen, joita ovat muun muassa ASP.NET internetsovelluksia varten ja ADO.NET tietokantayhteyksiin. .NET Frameworkia voidaan käyttää kolmen eri ohjelmointikielen kanssa. Näitä ovat C#-, F#- ja Visual Basic -ohjelmointikielet. [Microsoft 2019d]

4.3.2 .NET Core

.NET Core on pääasiassa Microsoftin kehittämä ohjelmistokehys, jonka avulla voidaan kehittää sovelluksia useammalle eri alustalle, muun muassa Windowsille, macOS:lle ja Linuxille. .NET Core on myös kokonaan avoimen lähdekoodin toteutus, toisin kuin .NET Framework. .NET Corella on samankaltaisuuksia .NET Frameworkin kanssa. .NET Corella kehitetyt ohjelmistot toimivat samalla periaatteella kuin .NET Frameworkillakin. C#, F# tai Visual Basic -ohjelmointikielellä kirjoitetut sovellukset käännetään ensin välikielimuotoisesti ja sovellus ajetaan CoreCLR:n avulla. CoreCLR on alustakohtainen. [Microsoft 2019e]

.NET Core poikkeaa .NET Frameworkista usealla eri tavalla. .NET Core ei tue kaikkia .NET Frameworkin sovellusmalleja, kuten ASP.NET MVC, mutta tukee ASP.NET Core MVC -sovellusmallia. .NET Core sisältää suuren osan .NET Frameworkin perusluokkakirjastosta, mutta erilaisilla perusasetelmilla. .NET Core toteuttaa osan alijärjestelmistä, joita on .NET Frameworkissa, sillä periaatteella, että kehittäminen ja ohjelmointimalli ovat yksinkertaisempia. .NET Corea voi käyttää useilla eri alustoilla, kun taas .NET Framework tukee vain Windows- ja Windows Server -käyttöjärjestelmiä. [Microsoft 2019e]

5 .NET ympäristöjen suorituskykyvertailu

5.1 Tutkimusmenetelmä

Tutkimus perustuu kahteen eri ohjelmaan, joista toinen on toteutettu käyttämällä .NET Framework 4.8:aa ja toinen käyttämällä .NET Core 3.0:aa. Ohjelmat toteuttavat saman toiminnallisuuden, joten ne on pyritty pitämään mahdollisimman identtisinä. Molemmissa ohjelmissa on toteutettu samat toiminnallisuudet ADO.NET:llä, Dapperilla ja Entity Frameworkilla. Joitain ohjelmistokehyksestä tai ORM-mallista riippuvaisia eroavaisuuksia ohjelmista löytyy.

Ohjelmat ajetaan erikokoisilla tietomäärillä. Näitä määriä ovat 10, 100, 1 000 ja 10 000 riviä. Toistokertoja jokaiselle ohjelman toiminnolle tulee olemaan kymmenen. Ohjelman suoritusajat kerätään aina jokaiselta toiminnolta. Eli kymmenellä toistokerralla saadaan kymmenen eri aikaa jokaiselle toiminnolle.

Tutkimuksessa keskitytään ORM-mallien aikatehokkuuteen, jolloin ei perehdytä tarkemmin ORM-mallien muuhun resurssien käyttöön. Aikatehokkuutta mitataan .NET-alustoihin sisältyvän Stopwatch-luokan avulla. Stopwatch-luokan avulla saadaan tarkkoja aikoja niiltä aikaväleiltä, joilta halutaan.

Tutkimuksessa ADO.NET-toteutus toimii perustason lähtökohtana, sillä se on .NET-ohjelmistokehyksien matalin taso toteuttaa tietokantayhteyksiä. Molemmat, Dapper ja Entity Framework, käyttävät ADO.NET-ohjelmistokehystä tietokantojen manipulointiin.

Tutkimuksessa tutkitaan sekä kirjoitus- että lukutehokkuutta. Jokaisen kirjoitus- ja lukutoiminnon jälkeen tietokanta tyhjennetään käyttämällä TRUNCATE TABLE -lauseketta. Tällöin rivien määrä taulussa pysyy johdonmukaisena, eikä edellisten toistokertojen tiedot vääristä tuloksia. Jokaisella toistokerralla suoritetaan vain yksi kirjoitustoiminto, jolla lisätään tietokantaan ennalta määritelty määrä rivejä ja yksi lukutoiminto, jolla luetaan yksi tietue tietokannasta.

5.2 Aiemmat tutkimukset

.NET-ohjelmistokehyksien ORM-malleja on vertailtu aikaisemminkin useammassa tutkimuksessa. Tutkimuksissa ei ole yleensä vertailtu kaikkia saatavilla olevia toteutuksia

ORM-malleista. ORM-mallien tehokkuutta on vertailtu yleensä käyttämällä kahta tai kolmea eri ORM-mallia tutkimuksessa. Kahta ORM-mallia ovat tutkimuksissaan käyttäneet Gruca ja Podsiadło [2014], Cvetković ja Janković [2010] ja Jones [2019]. Molemmat tutkimukset vertailivat Entity Frameworkia ja nHibernatea toisiinsa. Gruca ja Podsiadło ovat käyttäneet Entity Frameworkin versiota 5.0 ja nHibernaten versiota 3.3.1. Cvetković ja Janković käyttivät Entity Frameworkin .NET Framework 4.0:n mukana tullutta versiota ja nHibernaten 2.0.1 versiota. Jones vertaili keskenään Entity Framework Corea ja Dapperia. Gruca ja Podsiadło toteavat tutkimuksessaan, että nHibernate ja Entity Framework eivät merkittävästi eroa suorituskyvyltään. Cvetković ja Janković esittivät Entity Frameworkin suoriutuvan paremmin kuin nHibernate kaikissa, paitsi yhdessä testitapauksessa. Jonesin mukaan Dapper osoittautui huomattavasti nopeammaksi kuin Entity Framework Core.

Kolmea ORM-mallia ovat verranneet Zmaranda *et al.* [2020], Basheleishvili *et al.* [2019], Wiphusitphunpol ja Lertrusdachakul [2017] ja Jones [2015]. Zmaranda *et al.* vertailivat Entity Framework Core 2.2:sta, nHibernate 5.2.3:sta ja Dapper 1.50.5:sta. Basheleishvili *et al.* vertailivat keskenään ADO.NET:ä, Dapperia ja Entity Frameworkia. Wiphusitphunpol ja Lertrusdachakul keskittyivät ADO.NET 4.3.0, Dapper 1.50.3 beta ja Entity Framework Core 1.1.0 -versioihin. Jones vertaili keskenään Dapperia, Entity Frameworkia ja ADO.NET:ä. Zmaranda *et al.* tutkimuksessa ei löytynyt selkeää ehdokasta, jolla olisi ollut parhaat tulokset sekä suoritusajan että muistinkäytön näkökulmasta. Basheleishvili *et al.* toteavat tutkimuksessaan ADO.NET:n olleen nopein ja Entity Frameworkin hitain. Dapper ei poikennut tuloksiltaan merkittävästi ADO.NET:stä. Wiphusitphunpol ja Lertrusdachakul esittävät Entity Framework Coren sopivan sovelluksille, joille tietokannan rinnakkainen käyttöaste on pieni ja jotka toimivat ympäristössä, jossa on riittävästi muistia. Dapper on sopivampi sovelluksille, joille tietokannan rinnakkainen käyttöaste on suuri. Dapper on myös ADO.NET:ä parempi, koska se mahdollistaa tietokantakyselyjen vastauksien tuomisen suoraan olioihin. Jonesin mukaan Dapper on huomattavasti nopeampi kuin Entity Framework ja hieman nopeampi kuin ADO.NET.

Useimmiten tutkimuksissa on vertailtu Entity Frameworkia tai sen .NET Core -versiota Entity Framework Corea. Entity Frameworkia on verrattu nHibernateen, ADO.NET:iin ja Dapperiin. Entity Framework ja nHibernate ovat molemmat monipuolisempia toteutuksia ORM-mallista kuin Dapper, joka taas on hyvin kevyt toteutus ORM-mallista. Tässä tutkimuksessa perehdytään eri ORM-malleihin samoin kuin edellä mainituissakin

tutkimuksissa. Näitä ORM-malleja ovat Dapper ja Entity Framework. Näitä verrataan .NET:n tietokantayhteysohjelmistokehykseen ADO.NET:iin. Lisäksi tutkitaan myös, miten .NET-ohjelmistokehys vaikuttaa ORM-mallien tietokantakyselyjen suoritusaikoihin.

5.3 Ohjelman ajoympäristön tiedot

Ohjelmat ajetaan samalla tietokoneella, ja myös relaatiotietokanta on samalla tietokoneella. Tietokoneen fyysiset tiedot ovat:

- Käyttöjärjestelmä: Windows 10 Pro 64-bit
- Suoritin: Intel Core i5- 8600K @ 4600 MHz
- Muisti (RAM): 16,0 Gt
- Näytönohjain: Nvidia GeForce GTX 1070
- Kiintolevy: Kingston SA1000M8480G 250Gb

Tutkimuksen relaatiotietokantajärjestelmäksi on valittu Microsoft SQL Server. Valintaan on vaikuttanut paljon se, että sekä Microsoft SQL Server ja molemmat .NET-ohjelmistokehykset ovat Microsoftin tuotteita, ja näistä kaikista on tarjolla ilmainen versio, joita tässä tutkielmassa käytetään. Microsoft SQL Serverille on tehty hallintaohjelma Microsoft SQL Server Management Studio, joka tuo lisäarvoa tutkimuksen tekemiselle helpottamalla relaatiotietokantojen rakentamista, manipulointia ja ylläpitoa. Tutkimuksessa on käytetty Microsoft SQL Serverin kevyintä versiota Microsoft SQL Server 2017 Express -versiota. Kyseinen versio on ilmainen, helppo ottaa käyttöön ja samalla tarjoaa tarvittavan relaatiotietokantajärjestelmän tutkimuksen tekemistä varten.

Dapperin käytettävät versiot ovat .NET Frameworkin ohjelmistokehyksen kanssa 2.0.30 ja .NET Core ohjelmistokehyksen kanssa 2.0.30.

Entity Frameworkin versiot ovat .NET Frameworkin kanssa Entity Framework 6 (6.3.0) ja .NET Coren kanssa Entity Framework Core 3.0.0.

5.3.1 Ohjelmakoodi

Ohjelmat on kirjoitettu pääsiassa C#-ohjelmointikieltä käyttäen, lukuun ottamatta Dapper- ja ADO.NET-toteutuksien vaatimaa SQL-kielen käyttöä. Ohjelmista on luotu ajoversiot Microsoft Visual Studio Community 2019 -ohjelman avulla.

Ohjelmien toiminnallisuuteen kuuluu kolme toimintoa. Ensimmäiseen toimintoon kuuluu siemenluvun hakeminen relaatiotietokannasta ohjelmalle annetun SeedID-arvon avulla. Siemenluvun avulla generoidaan satunnaiset arvot kolmelle eri olion attribuutille RandomString, RandomDateTimeOffset ja RandomInt. Generoinnin jälkeen olio viedään relaatiotietokantaan. Toinen toiminto on se, että haetaan relaatiotietokannasta ensimmäisessä toiminnossa luoduista riveistä yksi, jolla on kaikista suurin arvo sarakkeessa RandomDateTimeOffset. Kolmas ja viimeinen toiminto on relaatiotaulun tyhjentäminen TRUNCATE TABLE -lausekkeen avulla. Tarkemmin ohjelmien toimintaan voi tutustua ohjelmakoodin avulla [Östman 2019a, 2019b].

Molemmille ohjelmille voidaan määritellä kolme erilaista muuttujaa, joiden mukaan ohjelma ajetaan. Ensimmäinen muuttuja on Rows, joka kuvaa tietokantaan vietävien rivien ja tietueiden määrää. Toinen muuttuja on SeedID, joka on halutun siemenluvun PRIMARY KEY Seed-taulussa. Kolmantena muuttujana on Repeats eli toistokertojen määrä. Toistokertojen määrällä tarkoitetaan sitä, kuinka monta kertaa ohjelman toiminnallisuus ajetaan. Yksi toistokerta sisältää siemenluvun haun, rivien generoinnin, rivien tietokantaan viennin, suurimman RandomDateTimeOffset-tiedon omaavan rivin haun ja relaatiotaulun tyhjentämisen.

Aikamittaus aloitetaan jokaisen funktion alussa, kun kunkin toiminnon metodia on ohjelmassa kutsuttu. Mittaus lopetetaan, kun funktio on päässyt loppuunsa. Erillinen aikamittaus suoritetaan myös jokaisen ORM-mallin kolmen eri toiminnon kokonaisajalle, joka sisältää ohjelmalle mahdollisesti määritellyt toistokerrat. Tällöin voidaan vertailla ORM-mallien kokonaisaikatehokkuuksia paremmin toisiansa vasten.

5.3.2 Tutkimukseen käytetty tietojoukko

Tutkimuksessa ei käytetä yleisesti jaossa olevaa tietojoukkoa. Ohjelmien vaatima tieto generoidaan osana ohjelman suoritusta. Generointi tapahtuu siemenluvun avulla, jotta jokaisella ajokerralla saadaan sama tietojoukko, mikä helpottaa tutkimuksen tulosten toistamista muualla. Tutkimuksessa siemenlukuna on ollut arvo 140 894. Generointiin voi tutustua paremmin ohjelmakoodin avulla [Östman 2019a, 2019b].

5.3.3 Relaatiotietokanta

Tutkimuksessa käytettävä relaatiotietokanta koostuu kahdesta taulusta: Seed ja RandomObject. Seed-taulu koostuu kahdesta sarakkeesta SeedID ja SeedValue. SeedID:n tietotyyppi on *int* ja SeedValue-sarakkeen tietotyyppi on myös *int*. Seed-taulussa SeedID on PRIMARY KEY, joka erottelee jokaisen rivin toisistaan. RandomObject-taulussa on viisi saraketta: RandomObjectID, RandomString, RandomDateTimeOffset, RandomInt ja SeedID. RandomObjectID on RandomObject-taulun PRIMARY KEY ja on tietotyypiltään *int*. RandomString on tietotyypiltään *string*, RandomInt on tietotyypiltään *int* ja RandomDateTimeOffset on tietotyypiltään *DateTimeOffset*. RandomObject-taulun SeedID-sarake on tietotyypiltään *int* ja on samalla FOREIGN KEY tauluun Seed. Microsoft SQL Management Studiolla generoidut SQL-lausekkeet, joilla voi luoda tutkimuksen mukaisen relaatiotietokannan, löytyvät liitteestä 1. Lausekkeita ovat taulujenluontilausekkeet tutkimuksen molemmille tauluille.

5.4 Tulokset

Tutkimuksen ohjelmien suoritusajat on taulukoitu liitteeseen 2. Taulukoita on 24 eli yksi jokaiselle ohjelmistokehyksen, ORM-mallin ja toistokertojen määrän yhdistelmälle. Taulukoissa ensimmäisessä sarakkeessa on toistokerran numero ja INSERT-sarakkeessa on suoritusajat alakohdassa 5.2.1 esitetylle ensimmäiselle toiminnolle. SELECT-sarakkeessa on toiselle toiminnolle ja viimeisessä TRUNCATE-sarakkeessa on kolmannelle toiminnolle.

5.4.1 .NET Framework

ORM / Rivi-määrä	10	100	1 000	10 000
ADO.NET	00:00.0553721	00:00.2265580	00:01.6878609	00:16.9006331
Dapper	00:00.0639377	00:00.2621456	00:01.7131246	00:18.1490532
Entity Framework	00:01.0849620	00:01.3341653	00:10.1719386	12:39.0206398

Taulukko 1: Ohjelman kokonaissuoritusaika kymmenellä toistokerralla .NET Frameworkilla.

Taulukko 1 kuvaa ohjelman suoritusaikoja jokaiselle ORM-mallille ja toistokertojen määrälle, kun ohjelman toteutuksessa on käytetty .NET Frameworkia. Ajat taulukossa ovat muodossa mm:ss.fffffff, eli mm kuvaa minuutteja, ss kuvaa sekunteja ja fffffff kuvaa sekunnin murto-osia. Ajan mittaus on aloitettu ennen funktiokutsuja INSERT-, SELECT- ja TRUNCATE-toiminnallisuuksille ja ajan mittaus on lopetettu näiden kolmen toiminnallisuuden suorituksen jälkeen.

5.4.2 .NET Core

ORM / Rivi-määrä	10	100	1 000	10 000
ADO.NET	00:00.2235219	00:00.3708459	00:01.8180038	00:15.0536514
Dapper	00:00.0789092	00:00.2249620	00:01.5875110	00:15.3685004
Entity Framework	00:00.6434433	00:00.6813297	00:03.0530232	00:24.5533130

Taulukko 2: Ohjelman kokonaissuoritusaika kymmenellä toistokerralla .NET Corella.

Taulukko 2 kuvaa ohjelman suoritusaikoja jokaiselle ORM-mallille ja toistokertojen määrälle, kun ohjelman toteutuksessa on käytetty .NET Corea. Muutoin taulukko 2 on koottu vastaavasti kuin taulukko 1.

6 Aikamittausten tulosten vertailu

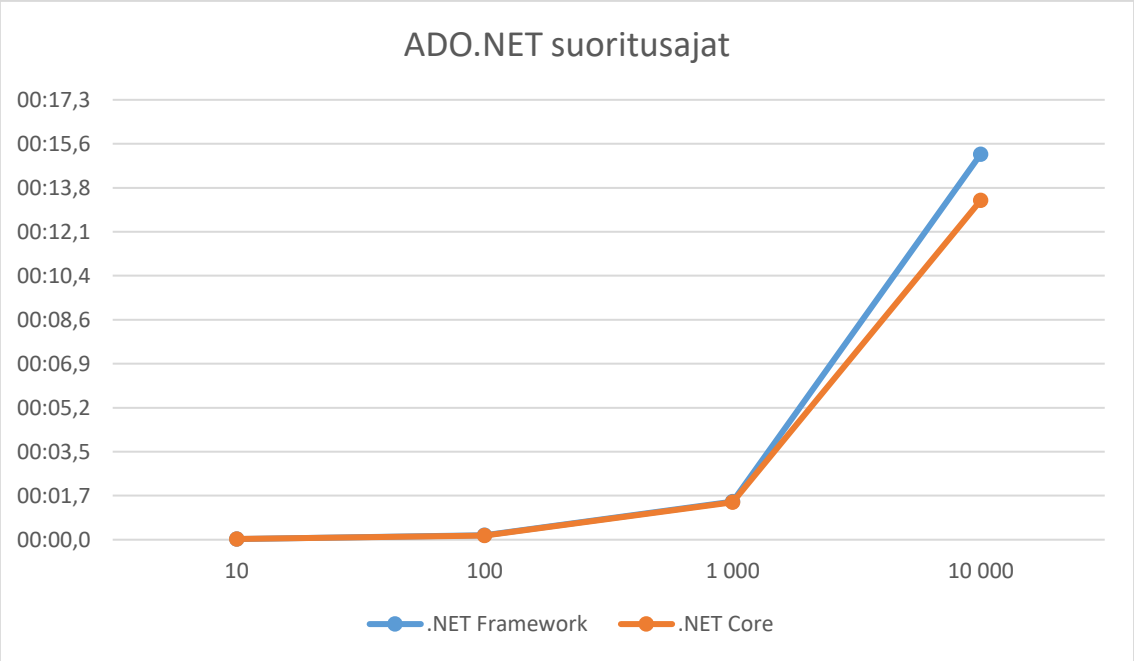
6.1 ADO.NET

Vertailemalla taulukkoja 1 ja 2 huomataan, että .NET Framework- ja .NET Core -toteutuksien kokonaissuoritusaikojen välillä on aikaeroja, paikoittain jopa huomattavia aikaeroja. ADO.NET on .NET Frameworkin kanssa .NET Corea nopeampi, kun rivien määrä on 10, 100 tai 1 000, mutta 10 000 rivillä .NET Core on nopeampi. Mutta jos katsotaan liitteen 3 taulukoita 1, 2, 3, 4, 13, 14, 15 ja 16, huomataan, että ensimmäisen toistokerran aika on aina suurempi kuin jälkimmäisten toistokertojen, varsinkin .NET Corea käytettäessä. Jos jätetään pois ensimmäinen pidempi suoritusaika, saadaan .NET Framework ja ADO.NET yhdistelmälle kokonaissuoritusaika 00:00.0253850 yhdeksällä toistolla ja 10 rivillä ja .NET Corea käyttämällä aika 00:00.0281966. Vastaavat ajat 100, 1 000 ja 10 000 rivimäärille ovat: 00:00.1761145 ja 00:00.1620858, 00:01.4896810 ja 00:01.4639903, 00:15.1448393 ja 00:13.3290779. Jättämällä poikkeavan ensimmäisen toiston pois, .NET Framework on nopeampi enää vain 10 rivillä ja .NET Core on nopeampi 100, 1 000, 10 000 rivillä.

Rivimäärä / .NET	.NET Framework	.NET Core	Ero %
10	00:00.0253850	00:00.0281966	+11,08
100	00:00.1761145	00:00.1620858	-7,97
1 000	00:01.4896810	00:01.4639903	-1,72
10 000	00:15.1448393	00:13.3290779	-11,99

Taulukko 3: ADO.NET suoritusajat .NET Frameworkilla ja .NET Corella ilman ensimmäistä poikkeavaa toistoa.

Taulukosta 3 nähdään, miten ADO.NET aikatehokkuus vaihtelee sen mukaan, mitä ohjelmistokehystä käytetään. Suurimmat aikaerot ovat 10 ja 10 000 rivin välillä, jolloin .NET Framework on 10 rivillä nopeampi kuin .NET Core ja 10 000 rivillä taas .NET Core on nopeampi kuin .NET Framework.



Kuva 2: ADO.NET suoritusajat.

Kuvasta 2 huomataan, että .NET Framework- ja .NET Core -toteutukset ADO.NET:stä seuraavat samanlaista kehitystä suoritusaikojen kasvussa. .NET Frameworkin suoritusajat kasvavat rivimäärän kertaantuessa kymmenellä seuraavasti: 594 %, 746 % ja 917 %. .NET Coren suoritusajat kasvavat rivimäärän kertaantuessa kymmenellä seuraavasti: 475 %, 803 % ja 810 %. .NET Frameworkilla pienemmillä rivimäärillä suoritusaika ei kasva samassa suhteessa kuin rivien määrä, mutta kasvu 1 000 rivistä 10 000 riviin on hyvin lähellä kymmenkertaistumista. .NET Corella suoritusajan kasvu 10 rivistä 100 riviin kasvu on 475 %, eli vähän yli puolet rivimäärän kasvusta. Suoritusajan kasvu 100 rivistä 1 000 ja 1 000 rivistä 10 000 riviin ovat hyvin lähellä toisiaan, eroa on vain noin 7 prosenttiyksikköä.

6.2 Dapper

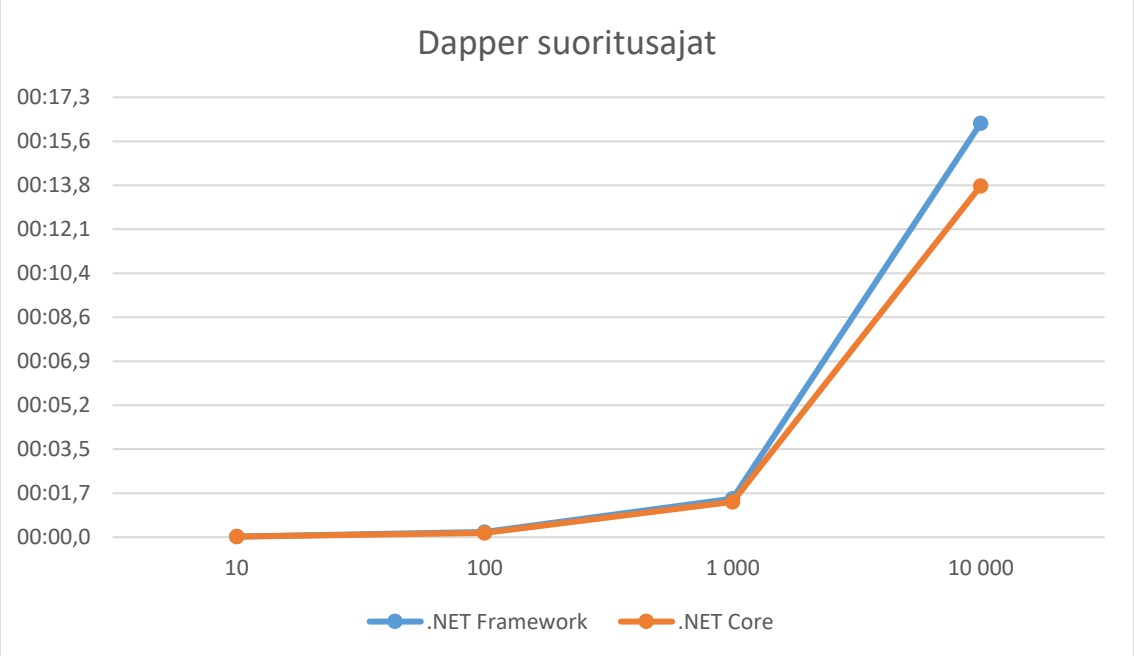
Dapperin suoritusajat .NET Frameworkilla ja .NET Corella nähdään myös taulukoista 1 ja 2. Dapper on nopeampi .NET Frameworkin kanssa 10 rivillä, mutta 100, 1 000 ja 10 000 rivillä .NET Core on nopeampi ohjelmistokehys. Katsomalla liitteen 2 taulukoita 5, 6, 7, 8, 17, 18, 19 ja 20 huomataan, että 10, 100, 1 000 rivimäärällä ensimmäinen toisto-

kerta on poikkeava muista toistokerroista. 10 000 rivillä INSERT-toiminnolla ei näy poikkeavuutta. Ensimmäisen toiston ollessa poikkeava muista toistoista myös Dapperia käytettäessä, jätetään se pois käsittelystä.

Rivimäärä / .NET	.NET Framework	.NET Core	Ero %
10	00:00.0241345	00:00.0228886	-5,16
100	00:00.2013308	00:00.1641469	-18,47
1 000	00:01.5092225	00:01.3807214	-8,51
10 000	00:16.2593988	00:13.8029284	-15,11

Taulukko 4: Dapper suoritusajat .NET Frameworkilla ja .NET Corella ilman ensimmäistä poikkeavaa toistoa.

Taulukosta 4 huomataan, että Dapper on .NET Corea käyttämällä nopeampi kuin .NET Frameworkilla. .NET Core on keskimäärin 11,81 prosenttia nopeampi kuin .NET Framework Dapperin käytössä. Suurin ero ohjelmistokehysten välillä on 100 rivillä, jolloin .NET Core on 18 prosenttia hitaampi, ja pienin ero 10 rivillä, jolloin .NET Core on vain 5 prosenttia hitaampi.



Kuva 3: Dapper suoritusajat.

Dapperin suoritusajat muuttuvat kuvan 3 mukaan hyvin samaan tapaan kuin ADO.NET:lläkin. Molemmat ohjelmistokehykset seuraavat samanlaista suoritusaikojen nousua. .NET Frameworkilla suoritusajat kasvavat rivimäärän kymmenkertaistuessa seuraavasti: 734 %, 650 % ja 977 %. .NET Corella vastaavat suoritusaikojen kasvut ovat: 617 %, 741 % ja 900 %. .NET Frameworkilla suoritusajan kasvu 10 rivistä 100 riviin ja 100 rivistä 1 000 riviin eivät kasva samassa suhteessa. Kasvu on suurempaa 10 rivistä 100 riviin, kuin kasvu 100 rivistä 1 000 riviin. Suoritusajan kasvu 1 000 rivistä 10 000 riviin on hieman suurempi kuin rivimäärien kasvu, noin 77 prosenttiyksikköä. .NET Core -toteutuksessa suoritusajan kasvu 10 rivistä 100 riviin ja 100 rivistä 1 000 riviin ei ole yhtä suuri kuin rivimäärien kasvu. Suoritusajan kasvu 1 000 rivistä 10 000 riviin on lähes täysin sama kuin rivimäärien kasvun määrä, suoritusajan kasvu on vain 0,3 prosenttiyksikköä pienempi kuin rivimäärän kasvu.

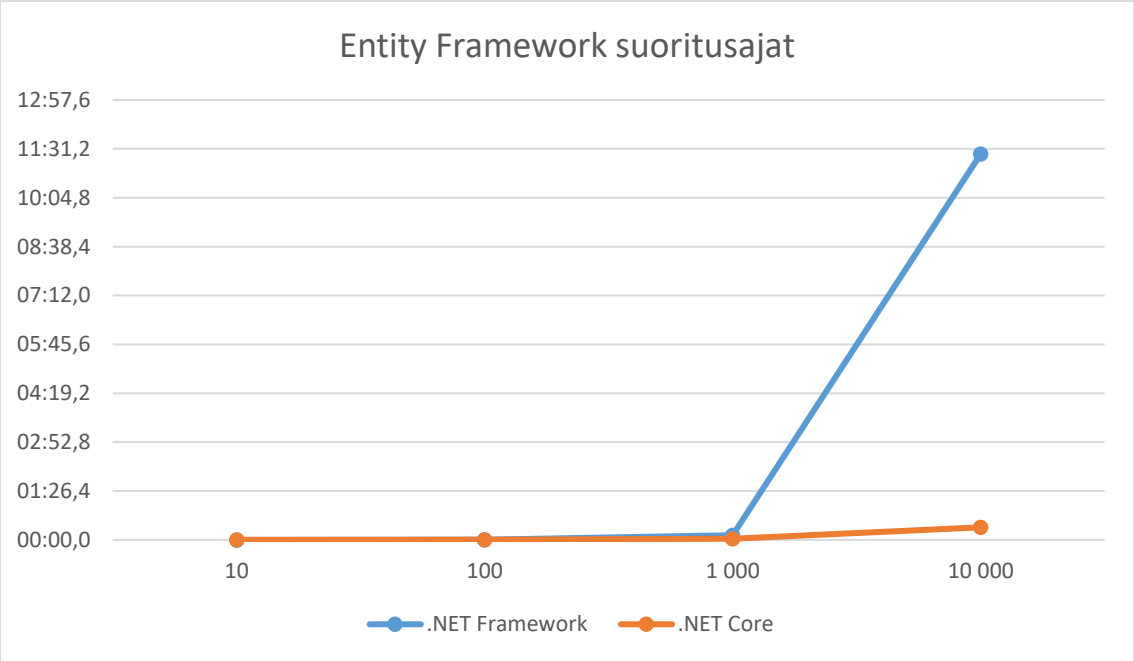
6.3 Entity Framework

Tarkastelemalla taulukoita 1 ja 2 huomataan, että Entity Framework on .NET Corella kaikilla rivimäärillä nopeampi kuin .NET Framework -toteutus. Suurin ero on 10 000 rivillä, kun .NET Frameworkin suoritusaika on noin 12 minuuttia ja 39 sekuntia ja .NET Coren suoritusaika on vain noin 25 sekuntia. .NET Framework on siis yli kolmekymmentä kertaa hitaampi kuin .NET Core 10 000 rivillä. Kuten ADO.NET- ja Dapper-toteutuksissa, myös Entity Frameworkia käyttämällä ensimmäinen toistokerta on suoritusajaltaan poikkeava kuin jälkimmäiset toistokerrat. Tämä voidaan todeta liitteen 2 taulukoista 9, 10, 11, 12, 21, 22, 23 ja 24. Jätetään ensimmäinen toistokerta pois käsittelystä.

Rivimäärä / .NET	.NET Framework	.NET Core	Ero %
10	00:00.0384654	00:00.0835470	+117,20 %
100	00:00.2537636	00:00.1215682	-52,09 %
1 000	00:08.2213316	00:02.0838089	-74,65 %
10 000	11:22.2287940	00:22.2293496	-96,74 %

Taulukko 5: Entity Framework suoritusajat .NET Frameworkilla ja .NET Corella ilman ensimmäistä poikkeavaa toistoa.

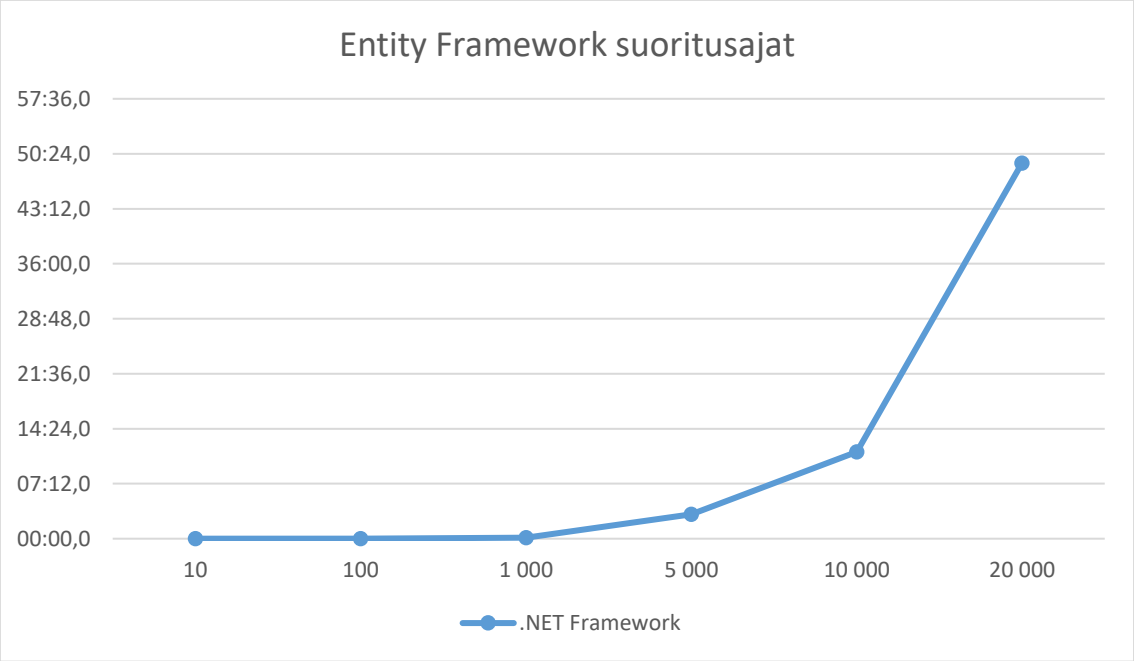
Taulukossa 5 nähdään suuria eroja Entity Frameworkin suoritusajoissa ohjelmistokehysten välillä. 10 rivillä .NET Framework on yli kaksi kertaa nopeampi kuin .NET Core. 100 rivillä asia on toisin päin eli .NET Core on hieman yli kaksi kertaa nopeampi kuin .NET Framework. 1 000 rivillä .NET Coren suoritusaika on enää vain noin yksi neljäsosa .NET Frameworkin suoritusajasta. 10 000 rivillä suoritusaikojen ero on kaikista suurin, sillä .NET Coren suoritusaika on vain noin 3,3 prosenttia .NET Frameworkin suoritusajasta.



Kuva 4: Entity Framework suoritusajat .NET Frameworkilla ja .NET Corella ilman ensimmäistä poikkeavaa toistoa.

Kuvasta 4 huomataan, että .NET Frameworkin ja .NET Coren suoritusajat eivät seuraa samanlaista kaavaa suoritusajan kasvussa. .NET Frameworkin suoritusaika lähtee suureen nousuun rivimäärään kasvaessa 1 000 rivistä 10 000 riviin, toisin kuin .NET Coren suoritusaika. .NET Frameworkia käytettäessä suoritusajat kasvavat rivimäärien kasvaessa kymmenkertaiseksi seuraavasti: 560 %, 3 140 % ja 8 200 %. Rivimäärän kasvaessa 10 rivistä 100 riviin suoritusaika ei kasva niin paljoa kuin rivimäärä. Kun rivimäärä kasvaa 100 rivistä 1 000 riviin, suoritusaika kasvaa huomattavasti enemmän kuin rivien määrä. Tällöin nähdään suoritusajassa yli 32-kertainen nousu. Siirryttäessä 1 000 rivistä 10 000 riviin suoritusaika kasvaa vielä enemmän kuin siirryttäessä 100 rivistä 1 000 riviin. Tällöin suoritusaika kasvaa melkein 83-kertaiseksi.

.NET Frameworkia ja Entity Frameworkia käytettäessä suoritusajat lähtevät suureen nousuun rivimäärää kasvatettaessa. Tämän takia .NET Frameworkia ja Entity Frameworkia käyttämällä tehtiin kaksi uutta ajoa. Uusiin ajoihin käytetyt rivimäärät ovat 5 000 ja 20 000. 5 000 rivillä suoritusaika 10 toistolla on 03:10.4332207 ja 20 000 rivillä suoritusaika on 49:11.5026384.



Kuva 5: Entity Framework suoritusajat .NET Frameworkilla ilman ensimmäistä poikkeavaa toistoa kuudella eri rivimäärällä.

Kuvan 5 avulla saadaan parempi käsitys, miten rivimäärän kasvu vaikuttaa suoritusajan kasvuun. Rivimäärän kasvaessa kaksinkertaiseksi 10 000 rivistä 20 000 riviin, suoritusaika kasvaa yli nelinkertaiseksi.

.NET Corea käyttämällä suoritusaikojen kasvut rivimäärien kymmenkertaistuessa eivät ole niin suuria kuin .NET Frameworkia käyttämällä. .NET Corella ne ovat 46 %, 1 610 % ja 967 %. .NET Coren suoritusajat heittelevät merkittävästi. Suoritusaika kasvaa rivimäärän kasvaessa 10 rivistä 100 riviin vain hieman alle puolella, kun taas 100 rivistä 1 000 riviin siirtyminen kasvattaa suoritusajan yli 17-kertaiseksi. 1 000 rivistä 10 000 riviin siirtyminen on lähellä rivimäärän kasvun määrää, eli kasvu on noin 67 prosenttiyksikköä enemmän kuin rivimäärän kasvu.

6.4 Ohjelmistokehykset

Tarkastelemalla taulukkoa 3 voidaan sanoa, että ADO.NET on nopeampi .NET Corea käyttämällä kuin .NET Frameworkia käyttämällä. .NET Framework on nopeampi vain 10 rivillä ja .NET Core on nopeampi 100, 1 000 ja 10 000 rivillä. .NET Core on keskimäärin 2,65 prosenttia nopeampi kuin .NET Framework. Prosentuaalisten suoritusaikaerojen mediaani on -4,85 prosenttia.

Taulukon 4 mukaan Dapper on nopeampi .NET Corella kaikilla rivimäärillä. .NET Core on keskimäärin 11,8 prosenttia nopeampi kuin .NET Framework. Prosentuaalisten suoritusaikaerojen mediaani on -11,8 prosenttia.

Entity Framework on taulukon 5 mukaan .NET Frameworkia käyttämällä selkeästi nopeampi kuin .NET Corea käyttämällä, kun rivimäärä on 10. Suuremmilla rivimäärillä .NET Core on huomattavasti .NET Frameworkia nopeampi. .NET Core on keskimäärin 26,6 prosenttia nopeampi kuin .NET Framework. Prosentuaalisten suoritusaikaerojen mediaani on -63,4 prosenttia.

Käytettäessä .NET Corea, nopein toteutustapa on ADO.NET taulukosta 6.1 lasketulla yhteissuoritusajalla 00:14.9833506. Toiseksi nopein on Dapper noin 2,5 prosenttia hitaammalla suoritusajalla 00:15.3706853. .NET Corea käyttämällä hitain toteutustapa on Entity Framework, jonka suoritusaika on 00:24.5182737, joka on noin 39 prosenttia hitaampi kuin ADO.NET-toteutuksen suoritusaika.

Käyttämällä .NET Frameworkia, nopeimmaksi toteutustavaksi osoittautui ADO.NET ajalla 00:16.8360178. Toiseksi nopein toteutus on Dapper yhteissuoritusajalla 00:17.9940865, joka on noin 6,5 prosenttia hitaampi kuin ADO.NET. Hitain toteutus on myös .NET Frameworkilla Entity Framework suoritusajalla 11:30.7423546, joka on noin 98 prosenttia hitaampi kuin ADO.NET.

7 Yhteenveto

Tutkielmassa käytiin läpi tietokantojen teoriaa ja historiaa sekä tutustuttiin paremmin relaatiotietokantoihin ja SQL-kieleen, minkä jälkeen keskityttiin olio-ohjelmoinnin ja relaatiotietokantojen välisiin *object-relational impedance mismatch* -ongelmiin ja sen kautta ORM-mallien syntyyn ja teoriaan. Lisäksi tutustuttiin myös ohjelmistokehyksiin ja niiden tuomiin etuihin ja haittoihin ohjelmistokehityksessä. Tarkemmin perehdyttiin .NET-ohjelmistokehyksiin .NET Framework ja .NET Core.

Luvussa 6 käytiin läpi, millaisia suoritusaikaeroja tulee .NET Frameworkin ja .NET Coren välille, kuten myös erillisten tietokantayhteyksien toteutustapojen ADO.NET:n, Dapperin ja Entity Frameworkin välille. .NET Corella tehdyt toteutukset osoittautuivat suoritusajoiltaan tehokkaimmiksi kuin .NET Frameworkilla. .NET Core oli nopeampi kymmenessä tapauksessa kahdestatoista. Poikkeuksena .NET Framework oli nopeampi ADO.NET toteutuksella 10 rivillä ja Entity Frameworkilla 10 rivillä, kuin .NET Core.

Entity Framework osoittautui kaikista kolmesta toteutustavasta hitaimmaksi. Pienemmillä rivimäärillä erot eivät ole suuria, mutta kun rivimäärää kasvatetaan, kasvaa myös Entity Frameworkin suoritusaikaerot muihin toteutuksiin verrattuna. Entity Framework tarjoaa ORM-mallina todella paljon toiminnallisuuksia verrattuna ADO.NET:iin ja Dapperiin, mutta nämä toiminnallisuudet näkyvät suoritusajoissa niiden suuruutena.

Rivimäärän kasvattaminen lisäsi ohjelman suoritusaikaa, mutta ei samassa suhteessa. Useimmissa tapauksissa suoritusaika ei kasvanut niin paljoa kuin rivien määrä. Dapperia ja ADO.NET:ä käyttämällä suoritusaika oli pienemmillä rivimäärillä alle kymmenkertainen, mutta 10 000 rivillä lähestyttiin samaa kasvusuhdetta kuin rivimäärällä. Entity Frameworkia käyttämällä esiintyi eniten poikkeavaisuuksia, jolloin suoritusaika saattoi kasvaa jopa 83-kertaiseksi, kun rivimäärä kasvoi vain 10-kertaiseksi. .NET Corea käyttämällä Entity Frameworkin ajat olivat samassa mittasuhteessa kuin ADO.NET:n ja Dapperin ajat, paitsi kasvatettaessa 10 rivistä 100 riviin, jolloin suoritusaika kasvoi vain 45,51 prosenttia. .NET Frameworkia käyttämällä rivimäärän kasvatus 100 rivistä 1 000 riviin ja 1 000 rivistä 10 000 riviin suoritusajan kasvu oli moninkertainen rivimäärän kasvuun verrattuna.

Tutkimusta on mahdollista jatkaa useammalla eri lähtökohdalla. Tutkimuksessa voisi käyttää jotain julkisesti saatavilla olevaa tietojoukkoa. Tällöin tutkimuksen tietojoukko

olisi lähempänä jokapäiväistä elämää kuin tässä tutkimuksessa käytetyt satunnaisgeneroidut arvot. Tutkimuksen voisi myös suorittaa käyttämällä eri siemenlukua kuin tässä tutkimuksessa, jolloin varmistuisi, onko siemenluvulla merkitystä suoritusaikoihin. Tutkimuksessa käytettävä datamäärä on rajoitettu 10 000 tietueeseen, joten isomman tietomäärän käyttäminen ja käyttämällä useampaa erilaista rivimäärää kuin tässä tutkimuksessa käytettyä neljää, olisi mahdollista saada kattavampi kuva tietomäärän vaikutuksesta suoritusaikaan.

Tässä tutkimuksessa Entity Framework -toteutuksien välillä ilmenee suuria suoritusaikaeroja. Ohjelmakoodia ei ole yritetty optimoida millään tavalla, vaan on käytetty suositeltua lähestymistapaa jokaiselle toteutustavalle. Optimoimalla toteutuksia olisi mahdollista saada tarkempi kuva siitä, miten toteutustavat poikkeavat suoritusajoiltaan toisistaan.

8 Viiteluettelo

Adya A., Blakeley J. A., Melnik S. and Muralidhar S. ADO.NET Team, Anatomy of the ADO.NET Entity Framework. *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. ACM, 2007. 877-888.

Atkinson M., Bancilhon F., DeWitt D., Dittrich K., Maier D. and Zdonik S. The object-oriented database system manifesto. *Deductive and Object-oriented Databases*. North-Holland, 1990. 223-240.

Basheleishvili I., Bardavelidze A. and Bardavelidze K. STUDY AND ANALYSIS OF THE .NET PLATFORM-BASED TECHNOLOGIES FOR WORKING WITH THE DATABASES. *Proceedings of the 33rd International Conference on Information Technologies (InfoTech-2019)*, Bulgaria. 2019. 1-8.

Bosch J, Molin P., Mattsson M. and Bengtsson P. Object-oriented framework-based software development: problems and experiences. *ACM Computing Surveys (CSUR)* 32.1es 2000. 3.

Butterfield A., Ngondi G. E. and Kerr A. eds. *A Dictionary of Computer Science*. Oxford University Press, 2016.

Cabibbo L. and Carosi A. Managing inheritance hierarchies in object/relational mapping tools. *International Conference on Advanced Information Systems Engineering*. Springer, Berlin, Heidelberg, 2005. 135-150.

Cattell R. G. G., Atwood T., Dubl J., Ferran G., Loomis M. and Wade D. *The Object Database Standard: ODMG*. Morgan Kaufmann Publishers, Los Altos, Calif, 1994.

Chamberlin D. D. and Boyce R. F. SEQUEL: A structured English query language. *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*. ACM, 1974. 249-264.

Codd E. F. A relational model of data for large shared data banks. *Communications of the ACM* 13.6 1970: 377-387.

Codd E. F. Relational database: a practical foundation for productivity. *Readings in Artificial Intelligence and Databases*. Morgan Kaufmann, 1989. 60-68.

Cvetković S. and Janković D. A comparative study of the features and performance of orm tools in a. net environment. *International Conference on Object and Databases*. Springer, Berlin, Heidelberg, 2010. 147-158.

Db-Engines, *DB-Engines Ranking*, *https://db-engines.com/en/ranking*, 2019a, tarkistettu 24.10.2019

Db-Engines, *DB-Engines Ranking - Trend Popularity*, *https://db-engines.com/en/ranking_trend*, 2019b, tarkistettu 3.11.2019

Db-Engines, *System Properties Comparison Microsoft SQL Server vs. MySQL vs. Oracle vs. PostgreSQL*, 2019c, *https://db-engines.com/en/system/Microsoft+SQL+Server%3BMySQL%3BOracle%3BPostgreSQL*, tarkistettu 3.11.2019

Deux O. The O2 system. *Communications of the ACM* 34.10 1991: 34-48.

Elmasri R. and Navathe S. B. *Fundamentals of Database Systems.* Benjamin/Cummings, Redwood City (Calif.), 1989.

Elmasri R. and Navathe S. B. *Fundamentals of Database Systems*. Vol. 7. Pearson, 2017.

Fayad M. and Schmidt D. C. Object-oriented application frameworks. *Communications of the ACM* 40.10 1997: 32-38.

GitHub, Dapper - a simple object mapper for .Net, 2019, *https://github.com/StackExchange/Dapper, tarkistettu* 21.11

Gray, J. The transaction concept: Virtues and limitations. *VLDB*. Vol. 81. 1981. 144-154.

Gruca A. and Podsiadło P. Performance Analysis of. NET Based Object–Relational Mapping Frameworks. *International Conference: Beyond Databases, Architectures and Structures*. 2014, Springer, Cham. 40-49.

Guru99, *C# and .Net Version History*, 2019, *https://www.guru99.com/c-sharp-dot-net-version-history.html*, tarkistettu 21.11.2019

Haerder T. and Reuter A. Principles of Transaction-oriented Database Recovery. *ACM Computing Surveys (CSUR)* 15.4 1983: 287-317.

Han J., Haihong E., Le G. and Du J. Survey on NoSQL database. *2011 6th International Conference on Pervasive Computing and Applications*. IEEE, 2011.

Harrington J. L. *Relational Database Design and Implementation: Clearly Explained.* Morgan Kaufmann/Elsevier, Amsterdam; Boston, 2009.

Ireland C., Bowers D., Newton M. and Waugh K. A classification of object-relational impedance mismatch. *2009 First International Conference on Advances in Databases, Knowledge, and Data Applications*. IEEE, 2009a, 36-43.

Ireland C., Bowers D., Newton M. and Waugh K. Understanding object-relational mapping: A framework based approach. *International Journal On Advances in Software* 2.2, 2009b, 202-216.

Jones M. Dapper vs Entity Framework vs ADO.NET Performance Benchmarking, 2015, *https://exceptionnotfound.net/dapper-vs-entity-framework-vs-ado-net-performance-benchmarking/*, tarkistettu 13.5.2020

Jones M. Dapper vs EF Core Query Performance Benchmarking, 2019, *https://exceptionnotfound.net/dapper-vs-entity-framework-core-query-performance-benchmarking-2019/*, tarkistettu 15.3.2020

Markiewicz M. E. and Lucena C. J. P. Object oriented framework development. *Crossroads* 7.4. 2001: 3-9.

Microsoft, *ADO.NET Tech Preview: Entity Data Model*, 2012, *https://docs.microsoft.com/en-us/previous-versions/aa697428(v=vs.80)?redirectedfrom=MSDN*, tarkistettu 21.11.2019

Microsoft, *Announcing .NET Core 1.0*, 2016, *https://devblogs.microsoft.com/dotnet/announcing-net-core-1-0/*, tarkistettu 21.11.2019

Microsoft, *ADO.NET Overview*, *https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview?redirectedfrom=MSDN*, 2017, tarkistettu 17.11.2019

Microsoft, *Past Releases of Entity Framework, https://docs.microsoft.com/en-us/ef/ef6/what-is-new/past-releases*, 2019a, tarkistettu 21.11.2019

Microsoft, What is .NET?, *https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet*, 2019b, tarkistettu 28.11.2019

Microsoft, What is .NET Framework? *https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet-framework*, 2019c, tarkistettu 4.12.2019

Microsoft, Get started with the .NET Framework., *https://docs.microsoft.com/fi-fi/dotnet/framework/get-started/index#Introducing*, 2019d, tarkistettu 4.12.2019

Microsoft, About .NET Core, *https://docs.microsoft.com/fi-fi/dotnet/core/about*, 2019e, tarkistettu 4.12.2019

NuGet, *Dapper*, *https://www.nuget.org/packages/Dapper/*, 2019, tarkistettu 21.11.2019

O'Neil E. J. Object/relational mapping 2008: hibernate and the entity data model (edm). *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. ACM, 2008, 1351-1356

Riehle D. *Framework Design: A Role Modeling Approach*. Diss. ETH Zurich, 2000.

Sammet J. E. The early history of COBOL. *History of Programming Languages I*. ACM, 1978.

Statista, *Global Digital Population as of October 2019*, *https://www.statista.com/statistics/617136/digital-population-worldwide/com/statistics/617136/digital-population-worldwide/*, 2019a, tarkistettu 24.10.2019

Statista, *Number of internet users worldwide from 2009 to 2019, by region*, *https://www.statista.com/statistics/265147/number-of-worldwide-internet-users-by-region/*, 2019b, tarkistettu 24.10.2019

Wiphusitphunpol W. and Lertrusdachakul T. Fetch performance comparison of object relational mapper in. NET platform. *2017 14th International Conference on Electrical Engineering/Electronics*, Computer, Telecommunications and Information Technology (ECTI-CON). IEEE, 2017. 423-426.

Zmaranda D., Pop-Fele L-L., Győrödi C., Győrödi R. and Pecherle G. Performance Comparison of CRUD Methods using NET Object Relational Mappers: A Case Study. *(IJACSA) International Journal of Advanced Computer Science and Applications,Vol. 11,* No.1, 2020, 55-65.

Östman T. FrameworkApp, 2019, *https://github.com/TuomasOstman/FrameworkApp*

Östman T. CoreDatabaseApp, 2019, *https://github.com/TuomasOstman/CoreDatabaseApp*

Liite 1 Tietokantataulujen luontilauseet

RandomObject:

```
CREATE TABLE [dbo].[RandomObject](
            [RandomObjectID] [int] NOT NULL,
            [RandomString] [nvarchar](max) NOT NULL,
            [RandomDateTimeOffset] [datetimeoffset](7) NOT NULL,
            [SeedID] [int] NOT NULL,
            [RandomInt] [int] NOT NULL,
 CONSTRAINT [PK_RandomObject] PRIMARY KEY CLUSTERED
(
            [RandomObjectID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, AL-
LOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO

ALTER TABLE [dbo].[RandomObject]  WITH CHECK ADD  CONSTRAINT [FK_Ran-
domObject_Seed] FOREIGN KEY([SeedID])
REFERENCES [dbo].[Seed] ([SeedID])
GO

ALTER TABLE [dbo].[RandomObject] CHECK CONSTRAINT [FK_RandomObject_Seed]
GO
```

Seed:

```
CREATE TABLE [dbo].[Seed](
            [SeedID] [int] NOT NULL,
            [SeedValue] [int] NOT NULL,
 CONSTRAINT [PK_Seed] PRIMARY KEY CLUSTERED
(
            [SeedID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, AL-
LOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

Liite 2 Suoritusajat toteutustavoittain

Taulukoiden 1 – 24 suoritusajat ovat mitattu jokaisen toiminnallisuuden funktion sisällä funktiokutsun jälkeen.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:00.0293685	00:00.0002096	00:00.0004090
2	00:00.0018615	00:00.0001349	00:00.0003092
3	00:00.0019183	00:00.0001342	00:00.0003199
4	00:00.0018166	00:00.0001312	00:00.0003089
5	00:00.0017753	00:00.0001281	00:00.0002917
6	00:00.0017994	00:00.0001306	00:00.0003002
7	00:00.0018304	00:00.0001295	00:00.0003599
8	00:00.0018342	00:00.0001353	00:00.0003063
9	00:00.0017876	00:00.0001302	00:00.0002976
10	00:00.0018688	00:00.0001289	00:00.0003022

Taulukko 1: 10 riviä 10 kertaa ADO.NETillä .NET Frameworkilla.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:00.0493247	00:00.0006967	00:00.0004221
2	00:00.0204573	00:00.0004885	00:00.0003958
3	00:00.0212227	00:00.0005023	00:00.0003759
4	00:00.0210778	00:00.0004992	00:00.0004191
5	00:00.0168036	00:00.0002333	00:00.0004859
6	00:00.0165909	00:00.0002196	00:00.0003403
7	00:00.0162621	00:00.0002132	00:00.0003310
8	00:00.0163168	00:00.0002377	00:00.0003659
9	00:00.0163745	00:00.0006749	00:00.0003697
10	00:00.0187254	00:00.0004983	00:00.0003934

Taulukko 2: 100 riviä 10 kertaan ADO.NETillä .NET Frameworkilla.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:00.1965389	00:00.0012229	00:00.0004181
2	00:00.1648333	00:00.0008338	00:00.0003502
3	00:00.1634736	00:00.0008652	00:00.0003420
4	00:00.1625638	00:00.0008232	00:00.0003441
5	00:00.1631019	00:00.0008732	00:00.0003533
6	00:00.1668887	00:00.0008389	00:00.0003499
7	00:00.1634665	00:00.0008732	00:00.0003471
8	00:00.1637035	00:00.0008650	00:00.0003478
9	00:00.1631255	00:00.0008600	00:00.0003425
10	00:00.1626072	00:00.0008215	00:00.0003436

Taulukko 3: 1 000 riviä 10 kertaa ADO.NETillä .NET Frameworkilla.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:01.7485938	00:00.0067283	00:00.0004717
2	00:01.6994849	00:00.0065540	00:00.0004575
3	00:01.6696883	00:00.0065227	00:00.0004080
4	00:01.6634527	00:00.0067262	00:00.0004459
5	00:01.6727563	00:00.0065531	00:00.0004492
6	00:01.6427277	00:00.0063900	00:00.0003788
7	00:01.6472003	00:00.0066066	00:00.0004092
8	00:01.6750834	00:00.0068367	00:00.0004420
9	00:01.7458313	00:00.0065381	00:00.0004248
10	00:01.6601900	00:00.0066028	00:00.0004444

Taulukko 4: 10 000 riviä 10 kertaa ADO.NETillä .NET Frameworkilla.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:00.0267768	00:00.0125851	00:00.0004413
2	00:00.0019085	00:00.0001480	00:00.0003065
3	00:00.0018440	00:00.0001304	00:00.0003060
4	00:00.0019124	00:00.0001391	00:00.0003242
5	00:00.0019740	00:00.0001369	00:00.0003113
6	00:00.0020085	00:00.0001560	00:00.0003403
7	00:00.0018823	00:00.0001453	00:00.0003139
8	00:00.0018219	00:00.0001393	00:00.0002993
9	00:00.0018456	00:00.0001322	00:00.0002998
10	00:00.0018064	00:00.0001278	00:00.0002962

Taulukko 5: 10 riviä 10 kertaa Dapperilla .NET Frameworkilla.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:00.0467601	00:00.0134217	00:00.0006333
2	00:00.0226337	00:00.0005405	00:00.0004025
3	00:00.0203078	00:00.0004736	00:00.0003724
4	00:00.0223572	00:00.0004758	00:00.0003784
5	00:00.0189877	00:00.0004949	00:00.0003784
6	00:00.0172007	00:00.0002183	00:00.0003473
7	00:00.0163704	00:00.0002137	00:00.0003395
8	00:00.0223452	00:00.0003163	00:00.0005005
9	00:00.0278186	00:00.0005537	00:00.0003910
10	00:00.0226939	00:00.0005471	00:00.0004177

Taulukko 6: 100 riviä 10 kertaa Dapperilla .NET Frameworkilla.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:00.1884944	00:00.0149410	00:00.0004667
2	00:00.1640037	00:00.0008497	00:00.0003549
3	00:00.1648852	00:00.0008234	00:00.0003541
4	00:00.1660337	00:00.0008225	00:00.0003524
5	00:00.1650265	00:00.0008235	00:00.0003500
6	00:00.1640554	00:00.0008152	00:00.0003647
7	00:00.1709568	00:00.0008212	00:00.0003489
8	00:00.1699068	00:00.0008600	00:00.0003561
9	00:00.1645996	00:00.0008667	00:00.0003490
10	00:00.1657682	00:00.0008670	00:00.0003514

Taulukko 7: 1 000 riviä 10 kertaa Dapperilla .NET Frameworkilla.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:01.8703424	00:00.0184865	00:00.0008255
2	00:01.9175738	00:00.0055073	00:00.0005353
3	00:01.7777858	00:00.0051174	00:00.0004228
4	00:01.6893780	00:00.0044701	00:00.0004154
5	00:01.8406521	00:00.0044034	00:00.0004070
6	00:01.6969009	00:00.0044056	00:00.0004272
7	00:01.6956188	00:00.0044420	00:00.0004203
8	00:01.7282084	00:00.0048775	00:00.0005028
9	00:01.8166338	00:00.0051306	00:00.0004896
10	00:02.0464045	00:00.0044702	00:00.0003848

Taulukko 8: 10 000 riviä 10 kertaa Dapperilla .NET Frameworkilla.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:01.0151897	00:00.0290160	00:00.0022909
2	00:00.0027869	00:00.0007827	00:00.0005440
3	00:00.0021939	00:00.0003827	00:00.0005087
4	00:00.0022077	00:00.0003780	00:00.0006256
5	00:00.0022123	00:00.0003832	00:00.0005003
6	00:00.0022258	00:00.0004577	00:00.0005660
7	00:00.0023185	00:00.0004016	00:00.0005175
8	00:00.0022443	00:00.0003801	00:00.0005506
9	00:00.0022086	00:00.0003904	00:00.0005136
10	00:00.0024408	00:00.0004660	00:00.0005589

Taulukko 9: 10 riviä 10 kertaa Entity Frameworkilla .NET Frameworkilla.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:01.0481125	00:00.0298799	00:00.0024093
2	00:00.0309206	00:00.0013229	00:00.0006259
3	00:00.0271217	00:00.0005995	00:00.0005869
4	00:00.0227470	00:00.0005645	00:00.0006876
5	00:00.0224888	00:00.0005652	00:00.0005567
6	00:00.0243417	00:00.0010214	00:00.0006201
7	00:00.0250992	00:00.0011770	00:00.0006273
8	00:00.0272012	00:00.0010265	00:00.0006034
9	00:00.0256844	00:00.0008552	00:00.0006221
10	00:00.0251123	00:00.0010838	00:00.0006232

Taulukko 10: 100 riviä 10 kertaa Entity Frameworkilla .NET Frameworkilla.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:01.9147237	00:00.0337058	00:00.0021775
2	00:00.9162209	00:00.0017670	00:00.0006713
3	00:00.9181676	00:00.0012406	00:00.0006868
4	00:00.9137762	00:00.0013071	00:00.0006086
5	00:00.9066916	00:00.0012625	00:00.0007262
6	00:00.9011014	00:00.0012694	00:00.0006021
7	00:00.9202135	00:00.0012852	00:00.0006610
8	00:00.9172174	00:00.0012867	00:00.0006723
9	00:00.8977034	00:00.0012618	00:00.0006045
10	00:00.9034949	00:00.0012890	00:00.0006047

Taulukko 11: 1 000 riviä 10 kertaa Entity Frameworkilla .NET Frameworkilla.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	01:16.7555777	00:00.0339983	00:00.0022698
2	01:16.4918812	00:00.0060114	00:00.0007799
3	01:15.7038579	00:00.0056400	00:00.0007783
4	01:15.5416777	00:00.0054796	00:00.0007647
5	01:16.0885521	00:00.0051858	00:00.0006884
6	01:15.7638922	00:00.0051854	00:00.0007104
7	01:15.7558524	00:00.0056776	00:00.0007610
8	01:15.3766368	00:00.0052679	00:00.0007011
9	01:15.5035409	00:00.0054173	00:00.0007555
10	01:15.9378640	00:00.0052810	00:00.0007050

Taulukko 12: 10 000 riviä 10 kertaa Entity Frameworkilla .NET Frameworkilla.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:00.1915141	00:00.0029100	00:00.0009012
2	00:00.0018029	00:00.0001497	00:00.0003034
3	00:00.0018741	00:00.0001518	00:00.0003258
4	00:00.0017860	00:00.0001299	00:00.0002976
5	00:00.0017388	00:00.0001474	00:00.0002878
6	00:00.0017649	00:00.0001225	00:00.0002891
7	00:00.0017632	00:00.0001210	00:00.0003004
8	00:00.0019838	00:00.0001324	00:00.0003292
9	00:00.0017810	00:00.0001408	00:00.0003563
10	00:00.0019923	00:00.0001412	00:00.0003169

Taulukko 13: 10 riviä 10 kertaa ADO.NETillä .NET Corella.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:00.2049655	00:00.0027911	00:00.0010035
2	00:00.0166596	00:00.0002125	00:00.0003449
3	00:00.0163779	00:00.0002143	00:00.0003345
4	00:00.0165411	00:00.0002156	00:00.0003336
5	00:00.0163519	00:00.0002210	00:00.0004465
6	00:00.0165252	00:00.0002131	00:00.0003346
7	00:00.0164745	00:00.0002232	00:00.0003350
8	00:00.0164577	00:00.0002272	00:00.0003420
9	00:00.0171310	00:00.0002164	00:00.0003392
10	00:00.0164743	00:00.0002198	00:00.0003377

Taulukko 14: 100 riviä 10 kertaa ADO.NETillä .NET Corella.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:00.3498950	00:00.0032173	00:00.0009012
2	00:00.1606895	00:00.0010131	00:00.0003939
3	00:00.1614313	00:00.0008611	00:00.0003949
4	00:00.1605676	00:00.0009872	00:00.0003635
5	00:00.1539235	00:00.0010166	00:00.0004417
6	00:00.1581329	00:00.0010581	00:00.0004162
7	00:00.1486264	00:00.0009714	00:00.0004147
8	00:00.1488326	00:00.0009497	00:00.0003682
9	00:00.2025083	00:00.0009943	00:00.0004736
10	00:00.1481601	00:00.0010391	00:00.0004139

Taulukko 15: 100 riviä 10 kertaa ADO.NETillä .NET Corella.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:01.7144560	00:00.0091520	00:00.0009655
2	00:01.4590482	00:00.0065565	00:00.0004122
3	00:01.4785363	00:00.0065399	00:00.0004474
4	00:01.4870017	00:00.0067100	00:00.0004521
5	00:01.4542272	00:00.0066406	00:00.0004542
6	00:01.4586857	00:00.0065662	00:00.0004447
7	00:01.5416225	00:00.0065080	00:00.0004851
8	00:01.4604619	00:00.0065536	00:00.0004349
9	00:01.4546479	00:00.0065660	00:00.0004230
10	00:01.4631922	00:00.0063249	00:00.0004004

Taulukko 16: 1 000 riviä 10 kertaa ADO.NETillä .NET Corella.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:00.0402203	00:00.0152600	00:00.0005403
2	00:00.0018811	00:00.0001681	00:00.0002990
3	00:00.0018693	00:00.0001476	00:00.0003116
4	00:00.0019159	00:00.0001547	00:00.0003269
5	00:00.0017733	00:00.0001339	00:00.0002929
6	00:00.0017959	00:00.0001309	00:00.0002911
7	00:00.0017728	00:00.0001322	00:00.0002906
8	00:00.0017648	00:00.0001304	00:00.0002907
9	00:00.0017715	00:00.0001282	00:00.0002927
10	00:00.0018272	00:00.0001379	00:00.0003029

Taulukko 17: 10 riviä 10 kertaa Dapperilla .NET Corella.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:00.0538952	00:00.0145140	00:00.0005023
2	00:00.0166932	00:00.0002445	00:00.0003540
3	00:00.0163106	00:00.0002149	00:00.0003395
4	00:00.0165461	00:00.0002251	00:00.0003478
5	00:00.0164346	00:00.0002164	00:00.0003375
6	00:00.0163695	00:00.0002120	00:00.0003343
7	00:00.0163765	00:00.0002125	00:00.0003472
8	00:00.0165091	00:00.0002045	00:00.0003367
9	00:00.0165539	00:00.0002079	00:00.0003303
10	00:00.0165580	00:00.0002095	00:00.0003371

Taulukko 18: 100 riviä 10 kertaa Dapperilla .NET Corella.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:00.1904707	00:00.0157635	00:00.0005554
2	00:00.1542147	00:00.0009683	00:00.0004034
3	00:00.1655857	00:00.0009293	00:00.0003779
4	00:00.1498904	00:00.0009243	00:00.0003793
5	00:00.1478168	00:00.0009683	00:00.0003823
6	00:00.1480678	00:00.0009192	00:00.0003794
7	00:00.1558273	00:00.0008791	00:00.0003446
8	00:00.1492087	00:00.0009448	00:00.0003643
9	00:00.1474545	00:00.0008973	00:00.0003631
10	00:00.1478602	00:00.0009008	00:00.0003462

Taulukko 19: 1 000 riviä 100 kertaa Dapperilla .NET Corella.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:01.5443238	00:00.0206944	00:00.0005538
2	00:01.4967346	00:00.0064641	00:00.0004097
3	00:01.6404229	00:00.0049091	00:00.0004917
4	00:01.6466020	00:00.0050548	00:00.0004677
5	00:01.5626705	00:00.0050631	00:00.0004833
6	00:01.4765837	00:00.0045031	00:00.0004065
7	00:01.4953809	00:00.0046851	00:00.0003744
8	00:01.4757064	00:00.0046494	00:00.0003685
9	00:01.4794562	00:00.0047018	00:00.0003741
10	00:01.4780445	00:00.0043571	00:00.0003762

Taulukko 20: 10 000 riviä 10 kertaa Dapperilla .NET Corella.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:00.5070083	00:00.0441340	00:00.0087540
2	00:00.0110842	00:00.0074079	00:00.0014622
3	00:00.0064779	00:00.0016665	00:00.0049057
4	00:00.0052462	00:00.0010387	00:00.0010195
5	00:00.0049926	00:00.0011903	00:00.0008469
6	00:00.0044463	00:00.0013664	00:00.0007819
7	00:00.0051158	00:00.0010860	00:00.0007172
8	00:00.0038836	00:00.0009162	00:00.0005901
9	00:00.0038404	00:00.0007813	00:00.0005217
10	00:00.0036095	00:00.0007892	00:00.0005471

Taulukko 21: 10 riviä 10 kertaa Entity Frameworkilla .NET Corella.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:00.5217697	00:00.0306074	00:00.0073844
2	00:00.0155165	00:00.0012929	00:00.0007253
3	0:00.0101588	00:00.0005164	00:00.0038515
4	00:00.0101268	00:00.0004775	00:00.0005145
5	00:00.0105040	00:00.0005758	00:00.0005605
6	00:00.0107911	00:00.0005212	00:00.0004471
7	00:00.0114084	00:00.0005243	00:00.0004665
8	00:00.0102726	00:00.0005030	00:00.0004674
9	00:00.0100240	00:00.0004544	00:00.0004208
10	00:00.0130340	00:00.0005126	00:00.0004441

Taulukko 22: 100 riviä 10 kertaa Entity Frameworkilla .NET Corella.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:00.9283239	00:00.0332165	00:00.0076739
2	00:00.2320185	00:00.0019576	00:00.0006381
3	00:00.2285184	00:00.0018274	00:00.0039544
4	00:00.2651658	00:00.0019404	00:00.0005486
5	00:00.2224736	00:00.0012358	00:00.0005112
6	00:00.2275977	00:00.0020299	00:00.0006043
7	00:00.2223065	00:00.0011688	00:00.0005215
8	00:00.2192667	00:00.0028708	00:00.0009615
9	00:00.2213468	00:00.0020222	00:00.0007270
10	00:00.2123905	00:00.0017675	00:00.0006925

Taulukko 23: 1 000 riviä 10 kertaa Entity Frameworkilla .NET Corella.

Toisto/Operaatio	INSERT	SELECT	TRUNCATE
1	00:03.2808127	00:00.0357086	00:00.0074421
2	00:02.4398743	00:00.0062908	00:00.0007952
3	00:02.3843374	00:00.0066922	00:00.0044518
4	00:02.3240134	00:00.0057854	00:00.0008181
5	00:02.3451305	00:00.0067437	00:00.0007904
6	00:02.3416324	00:00.0053201	00:00.0007779
7	00:02.3367613	00:00.0052882	00:00.0007856
8	00:02.3218453	00:00.0054609	00:00.0008144
9	00:02.3383047	00:00.0050642	00:00.0006962
10	00:02.3280994	00:00.0049877	00:00.0006749

Taulukko 24: 10 000 riviä 10 kertaa Entity Frameworkilla .NET Corella.